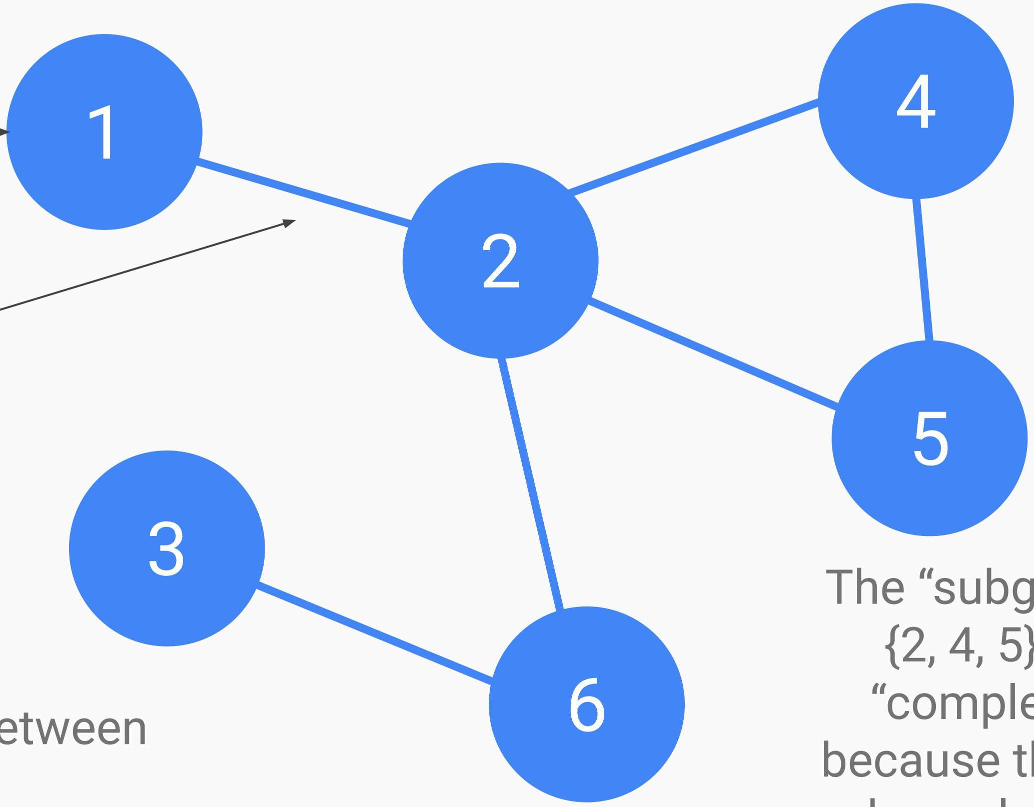
Trees and graphs

Code: https://github.com/mesmere/sasha-tutorial/tree/main/2024-02-03

Undirected graphs - Vocabulary



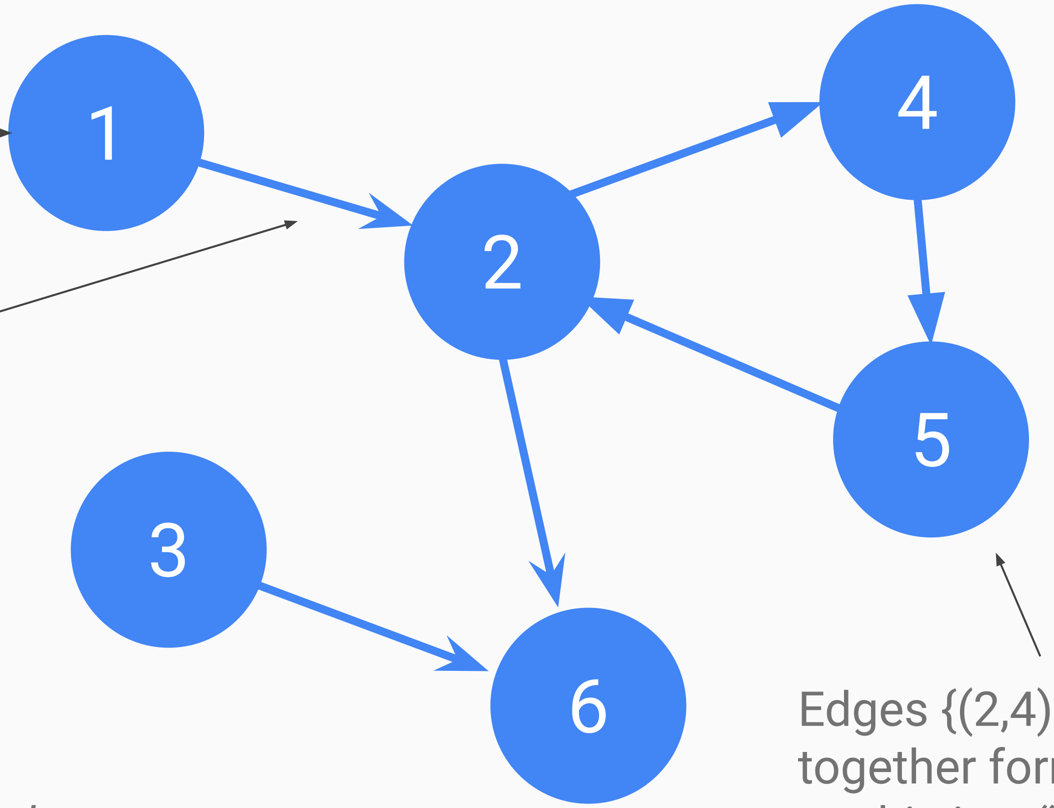
This graph is:
“Planar” = fits in a plane with
no intersecting edges
“Connected” = a path exists between
any two vertices
“Undirected” = edges go both ways

The “subgraph” {2, 4, 5} is “complete” because there’s nowhere else to put an edge.

Directed graphs - Vocabulary

“Vertex”

“Edge”



“Digraph”

=

the edges are *directed*.

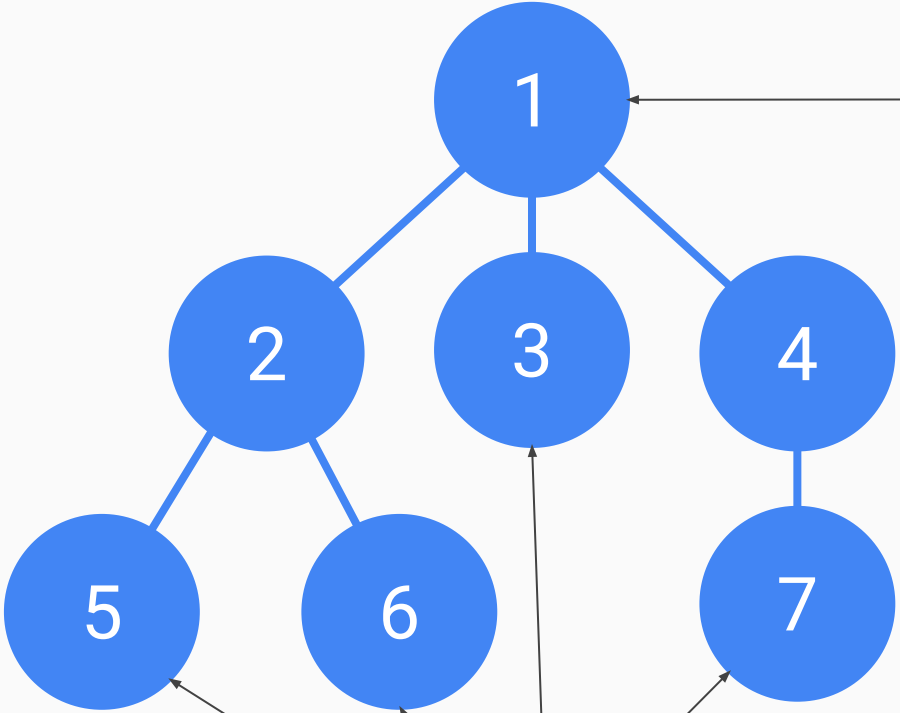
Edges {(2,4), (4,5), (5,2)} together form a “cycle” so this is a “cyclic graph.”

Trees - Vocabulary

A “tree” is a graph which is:

- Undirected
- Connected
- Acyclic

We’ll always pick one “root” node and think of edges as being directed with “parents” and (ordered) “children.”



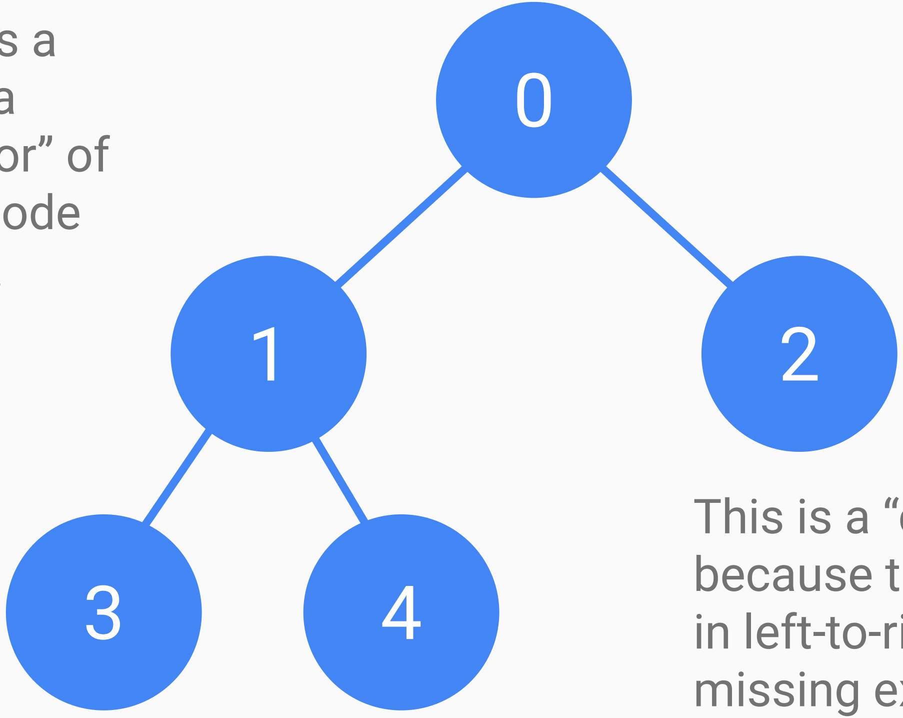
The root has no parent node. Every other node has exactly one parent.

A connected subgraph of a tree is a “subtree.”

“Leaves” have no child nodes.

Trees - Vocabulary

A “binary tree” is a tree which has a “branching factor” of two - i.e. each node has ≤2 children.



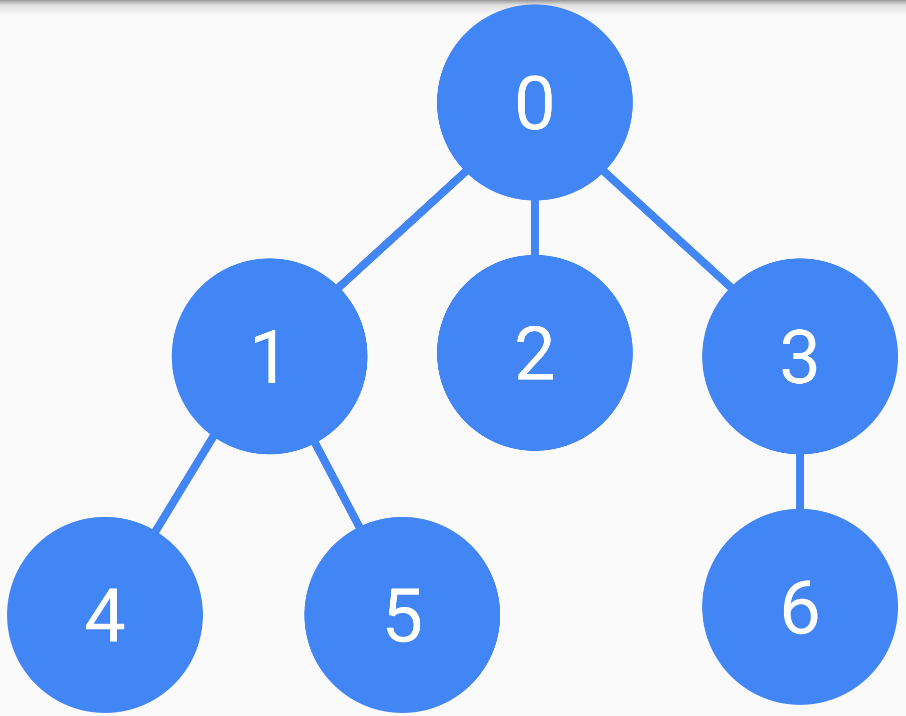
This is a “complete” binary tree because the levels have been filled in left-to-right with no spaces missing except at the end.

(A different meaning from the term “complete graph!”)

Trees

Tree representation

```
class Node {
  label;
  children;

  constructor(label, children = []) {
    this.label = label;
    this.children = children;
  }
}

const root = new Node("1", [
    new Node("2", [
        new Node("5"),
        new Node("6")
    ]),
    new Node("3"),
    new Node("4", [
        new Node("7")
    ])
]);
```

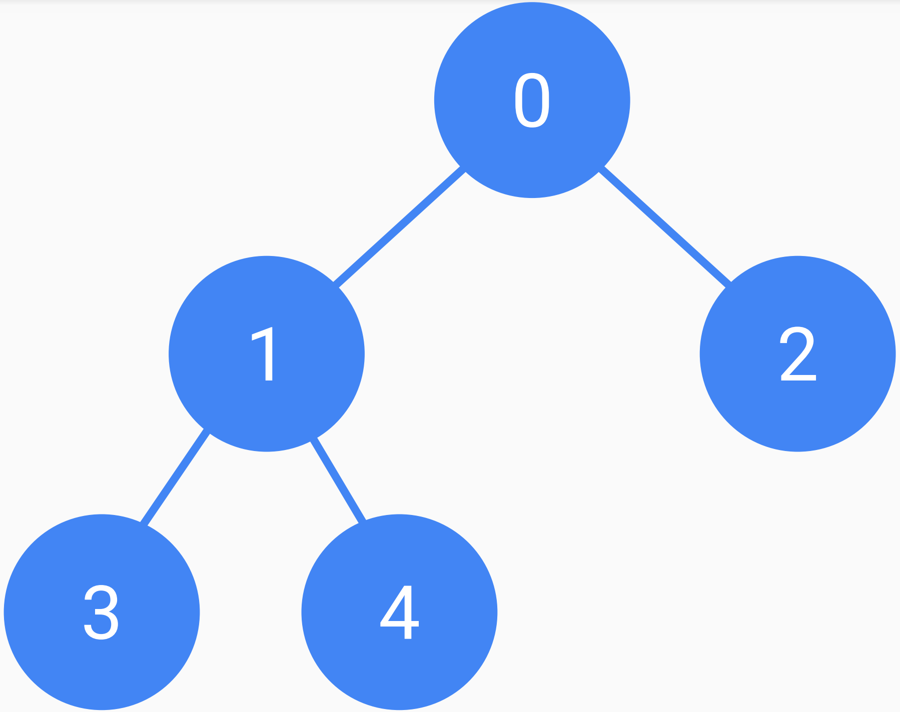


Binary tree representation 🤓

Interleave the nodes into **a single array** with the following scheme:

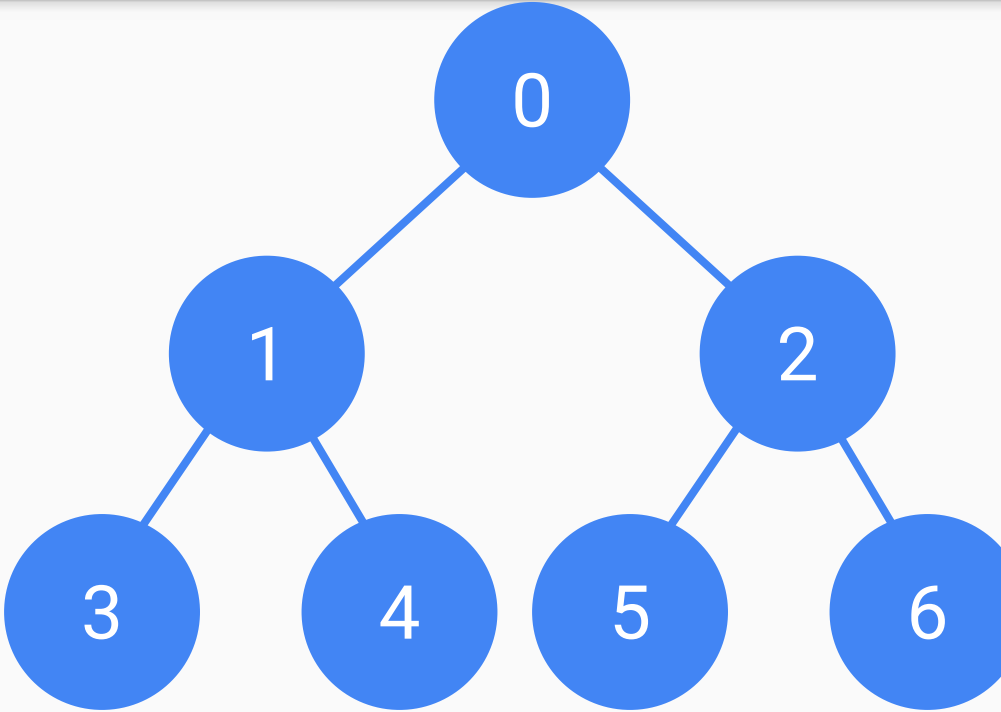
> Node 0, then
> Both children of node 0, then
> Both children of node 1, then
> Both children of node 2, then
> Both children of node 3, etc…

Leave `null` pointers if any spots in the complete binary tree are missing.



Binary tree representation 🤓

```
class BinaryTree {
  labels;

  constructor(labels = []) {
    this.labels = labels;
  }

  parentIndex(index) {
    if (index === 0) {
      throw "Already at the root.";
    }
    return Math.floor((index - 1) / 2);
  }

  leftChildIndex(index) {
    return (index * 2) + 1;
  }

  rightChildIndex(index) {
    return (index * 2) + 2;
  }
}

const tree = new BinaryTree(["1", "2", "3", "4", "5", "6"]);
```



Application - Heaps

“Priority queue” supported operations:

- Insert an element with some priority.
- Remove the element with the *lowest* priority.

How do we implement such a data structure **efficiently**?

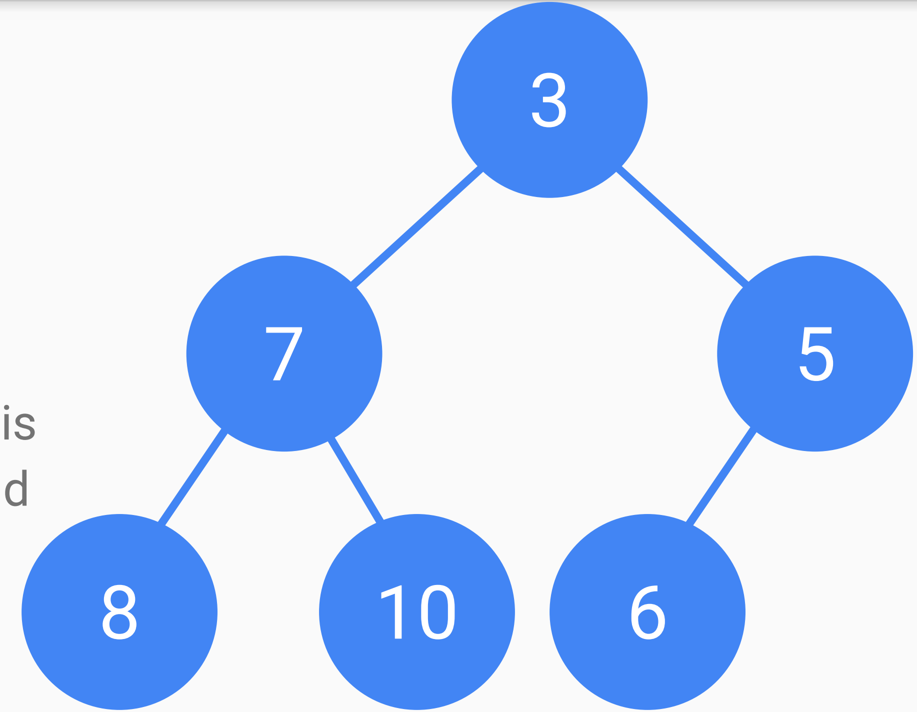
Application - Heaps

We can implement a priority queue using a heap.

Definition:

> A **binary minheap** is a complete binary tree such that every node is *less than or equal to* all of its child nodes.

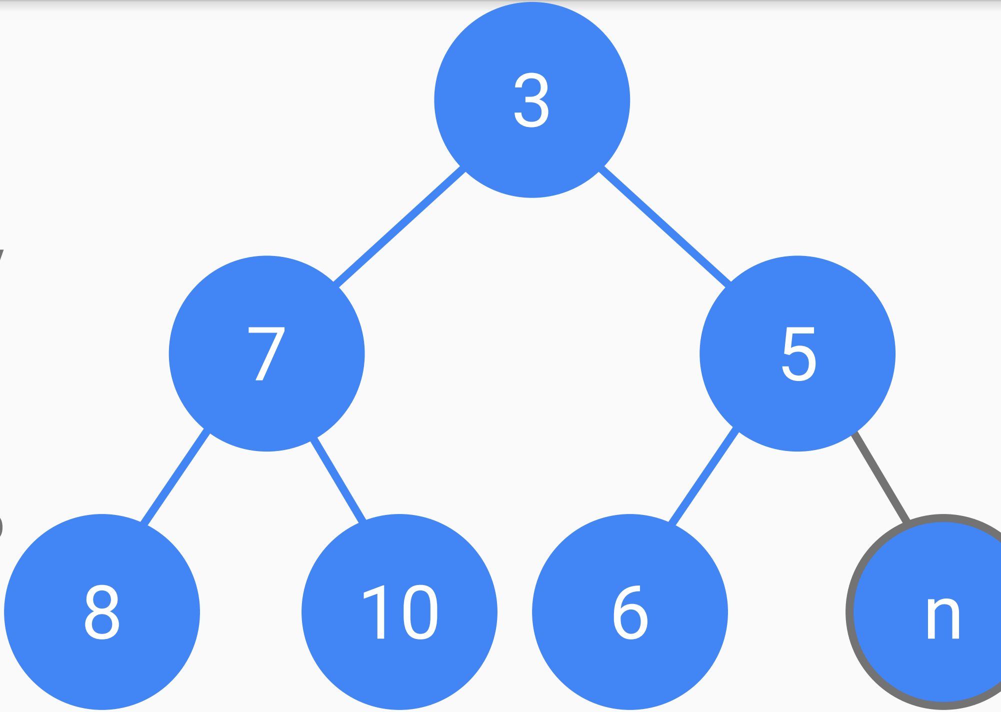
(This implies transitivity…)



Application - Heaps (insert operation)

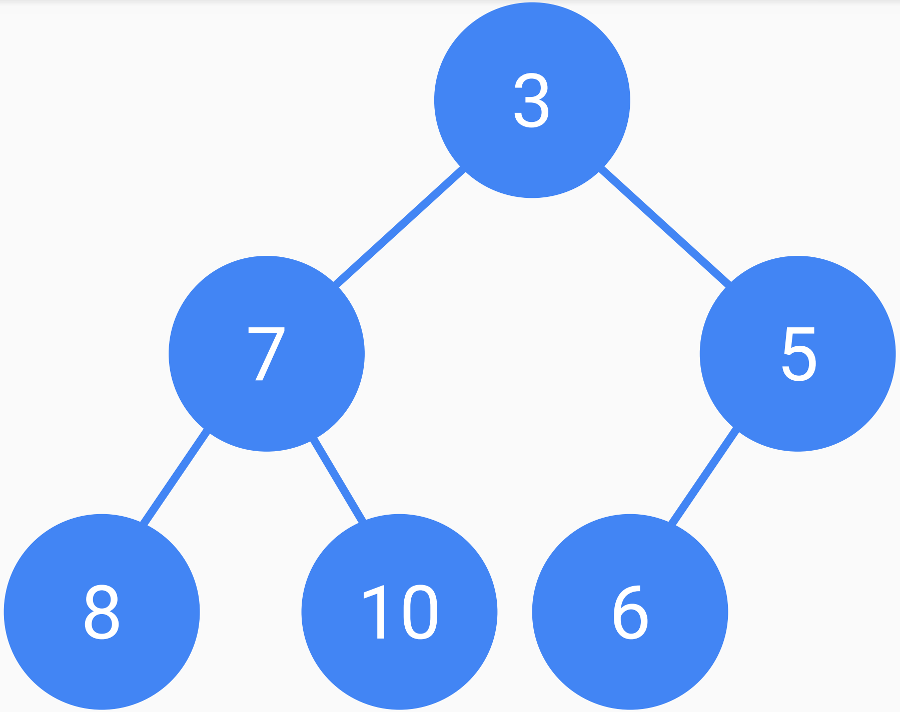
To insert a new element *n*:

1. Make a new node *n* at the "next spot" to keep it a complete binary tree. This may break the heap property!
2. Swap *n* with its parent as many times as necessary until the heap property is satisfied again.



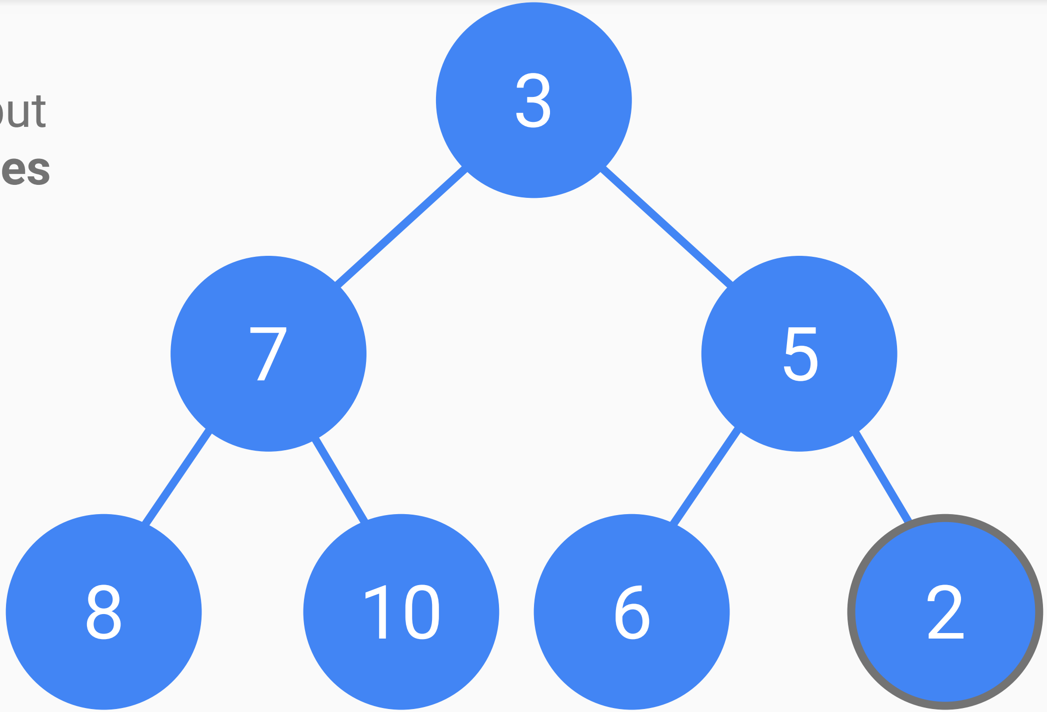
Application - Heaps (insert operation)

Let’s insert 2…

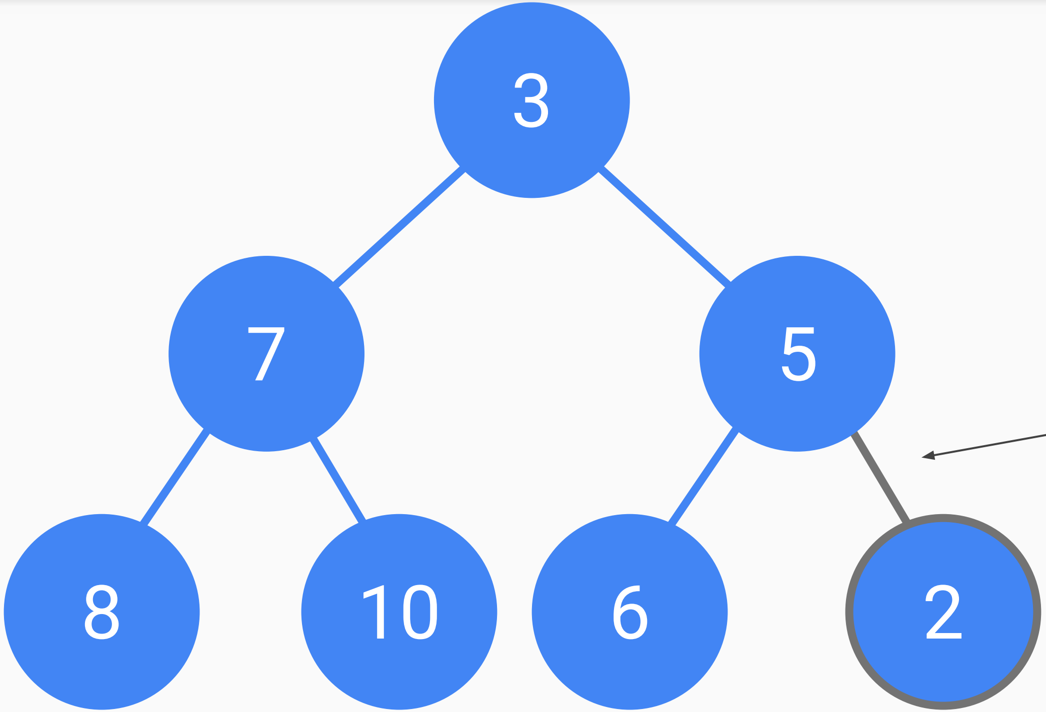


Application - Heaps (insert operation)

We've added 2 but now this tree **does not** satisfy the heap property because $5 \nleqslant 2$.

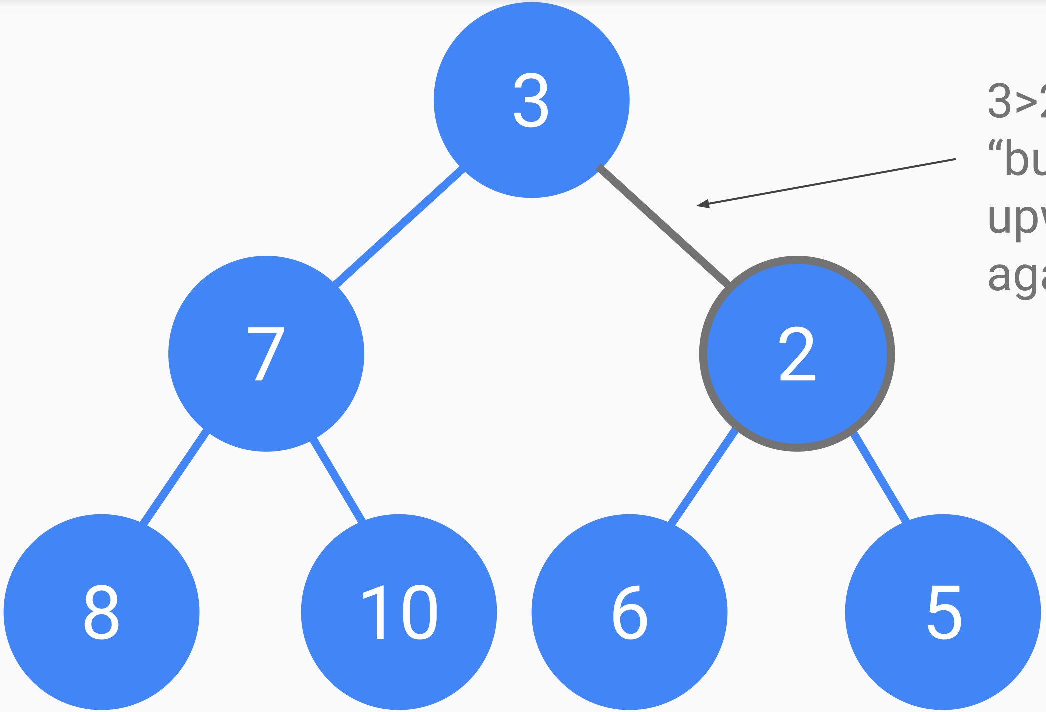


Application - Heaps (insert operation)



5>2 so we “bubble” 2 upwards.

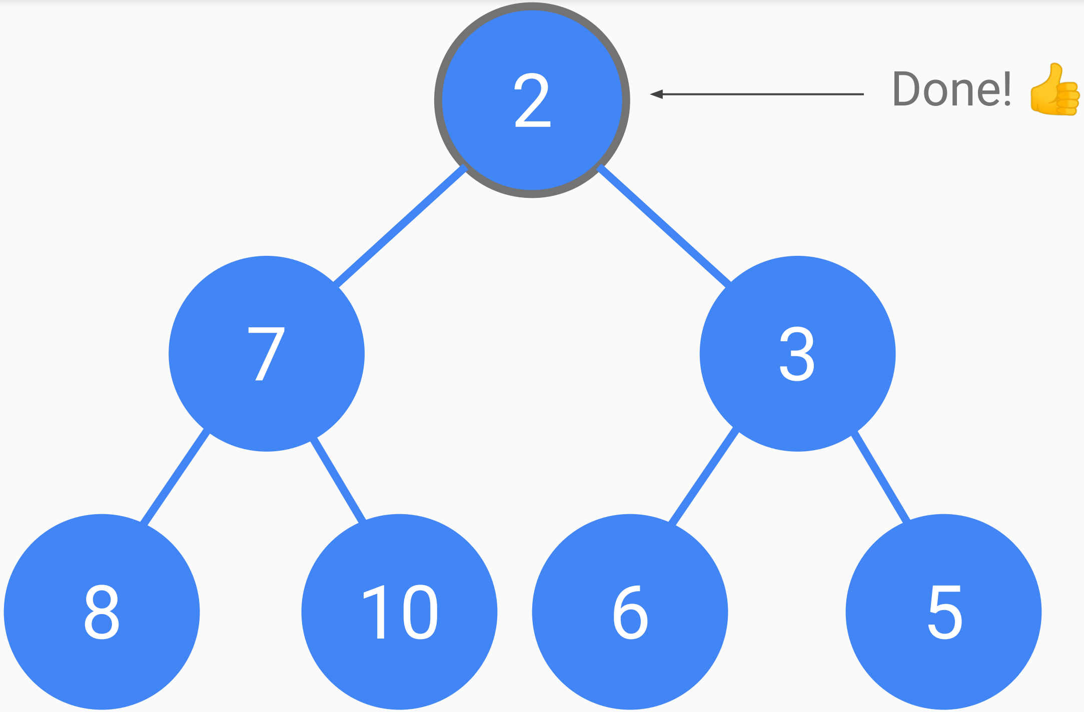
Application - Heaps (insert operation)



3>2 so we "bubble" 2 upwards again.

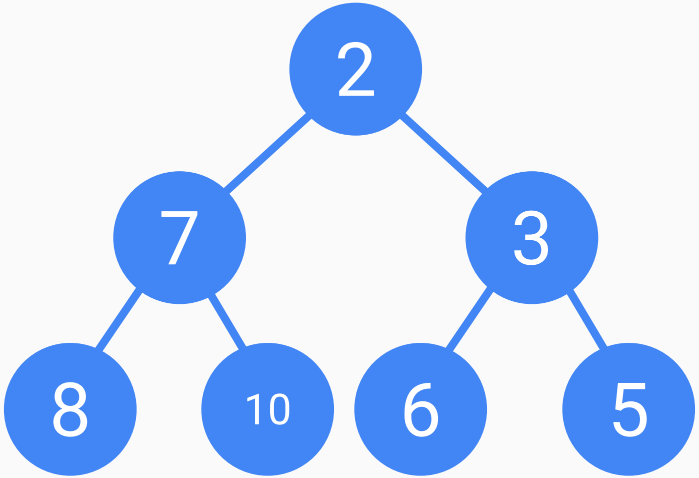
Application - Heaps (insert operation)

Convince yourself that this tree now satisfies the heap property.



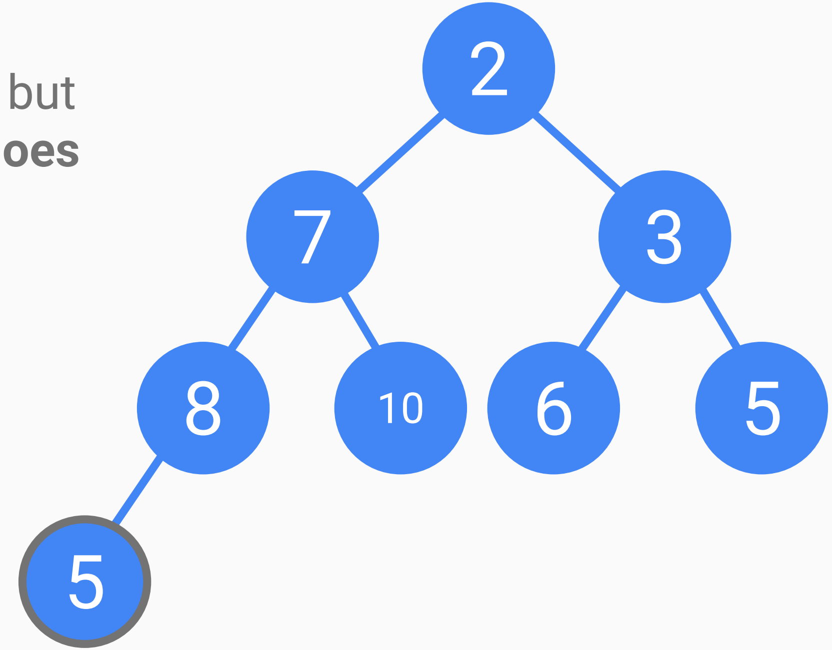
Application - Heaps (insert operation)

Let's insert 5 now...

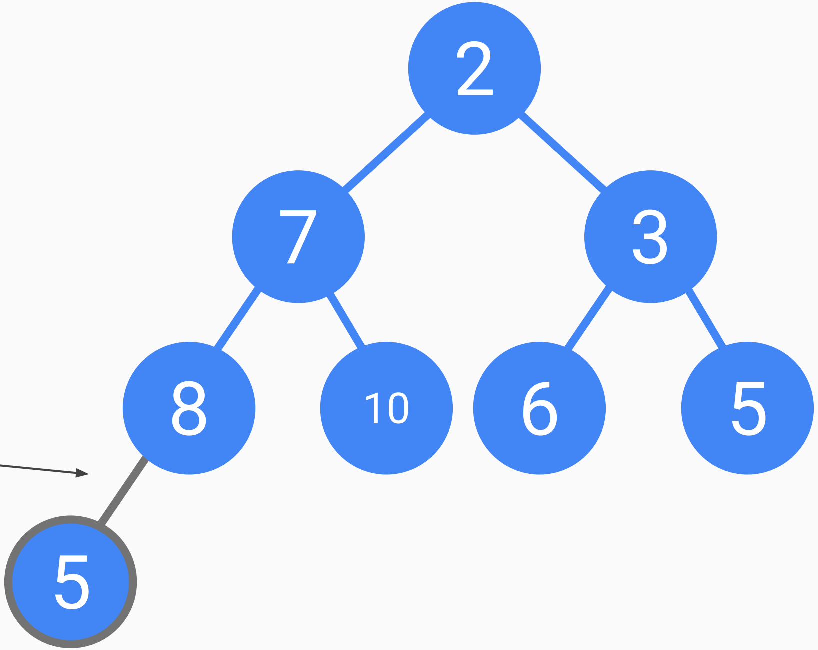


Application - Heaps (insert operation)

We've added 5 but now this tree **does not** satisfy the heap property because $8 \nleqslant 5$.



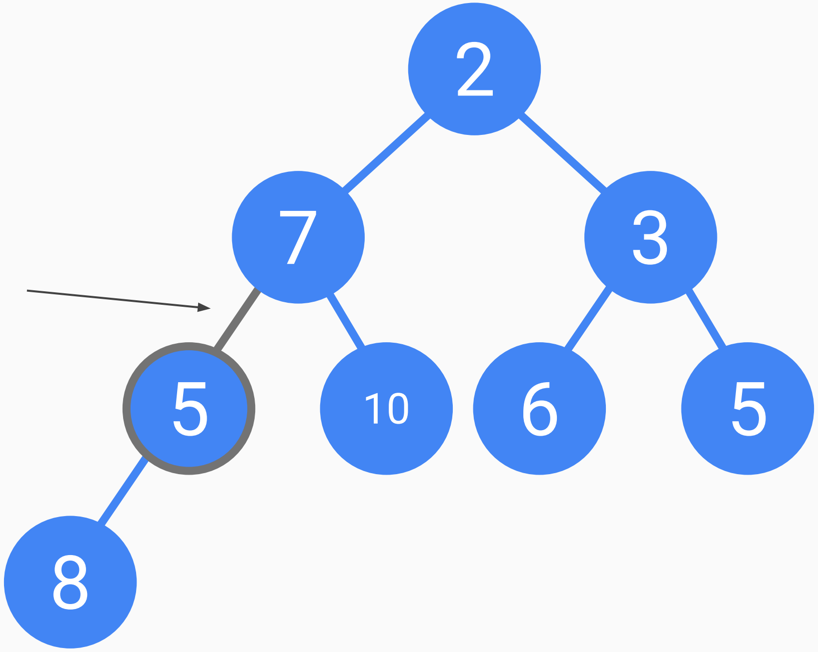
Application - Heaps (insert operation)



8>5 so we need to bubble up.

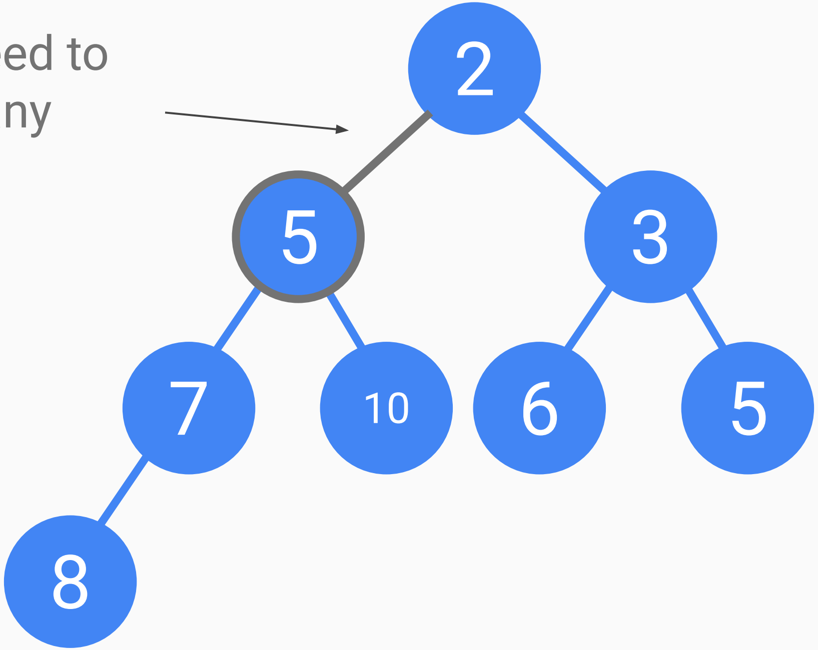
Application - Heaps (insert operation)

7>5 so we need to bubble up again.



Application - Heaps (insert operation)

We don’t need to bubble up any further. 👍



Convince yourself that this tree satisfies the heap property.

Application - Heaps (insert operation)

```
class PriorityQueue {
  items = [];

  #parentIndex(index) {
    return Math.floor((index - 1) / 2);
  }

  insert(item, priority) {
    let index = this.items.push({
      item: item,
      priority: priority
    }) - 1;
    while (index > 0
          && this.items[this.#parentIndex(index)].priority > this.items[index].priority) {
      // Swap this node with its parent.
      const oldParent = this.items[this.#parentIndex(index)];
      this.items[this.#parentIndex(index)] = this.items[index];
      this.items[index] = oldParent;

      index = this.#parentIndex(index);
    }
  }
}
```

Application - Heaps (insert operation)

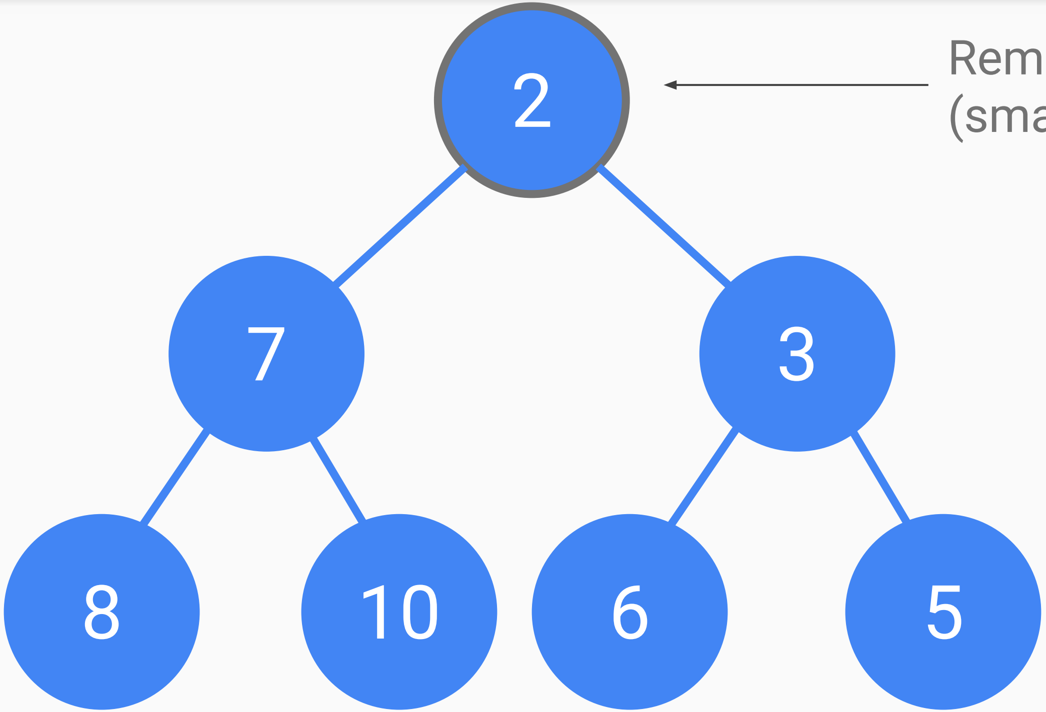
What is the worst-case running time of the insert operation (n = the heap size)?

Application - Heaps (remove operation)

To remove the element with the lowest priority:

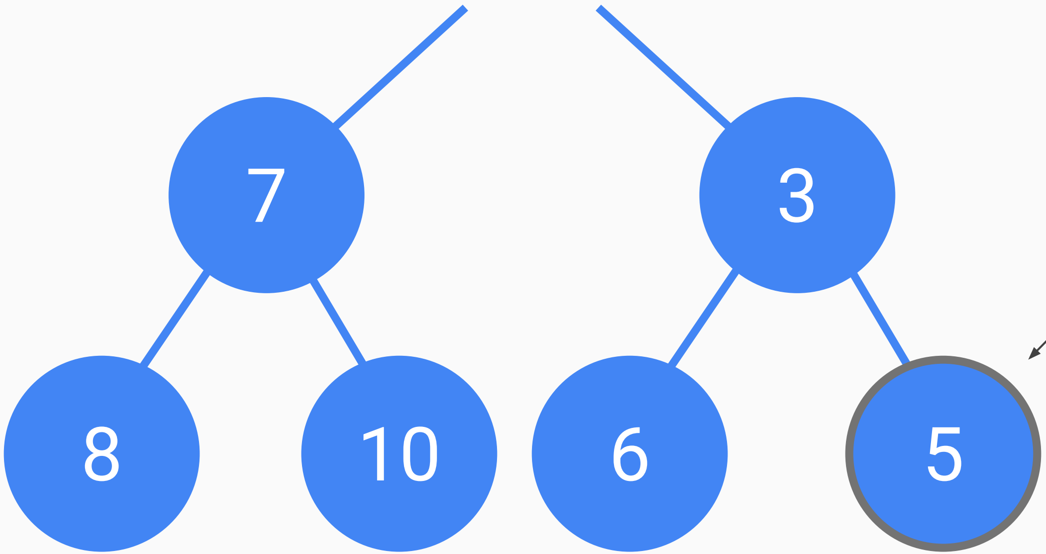
1. Remove the root node (it’s guaranteed to have the lowest priority).
2. Fill the empty space with the last node in the tree.
3. Repeatedly swap with the smallest child to restore the heap property.

Application - Heaps (remove operation)



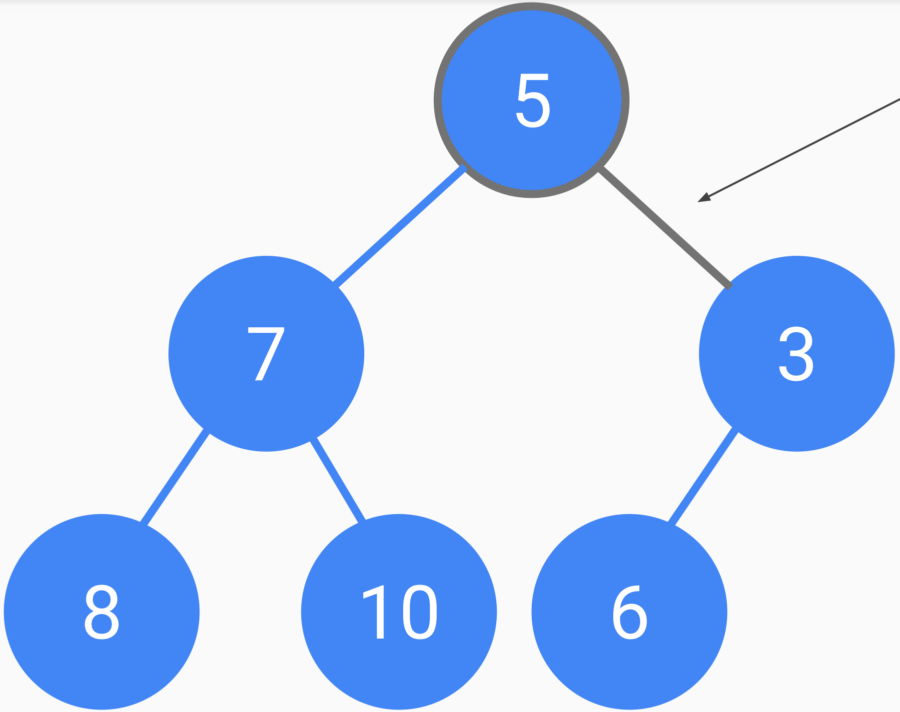
Remove the root
(smallest node)

Application - Heaps (remove operation)



Fill the space with the last node.

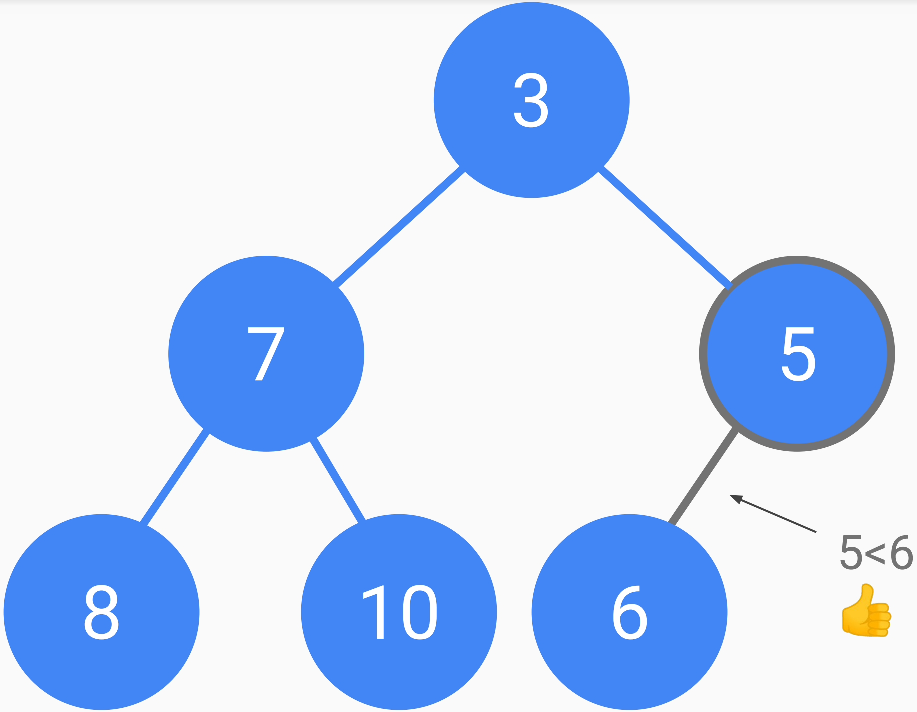
Application - Heaps (remove operation)



7<5 || 3<5
so swap with the
smallest child.

Application - Heaps (remove operation)

Convince yourself that this tree satisfies the heap property.



5<6 so we’re done. 👍

Application - Heaps (remove operation)

```
class PriorityQueue {
  items = [];

  #leftChildIndex(index) {
    return (index * 2) + 1;
  }

  #rightChildIndex(index) {
    return (index * 2) + 2;
  }

```

```
  remove() {
    if (this.items.length === 0) {
      return undefined;
    }

    if (this.items.length === 1) {
      return this.items.pop();
    }

    // Replace the root.
    const oldRoot = this.items[0];
    this.items[0] = this.items.pop();

    // Bubble down.
    let index = 0;
    while ((this.#leftChildIndex(index) < this.items.length
        && this.items[this.#leftChildIndex(index)].priority < this.items[index].priority
        ) || (
        this.#rightChildIndex(index) < this.items.length
        && this.items[this.#rightChildIndex(index)].priority < this.items[index].priority)) {
      // Figure out which child is the smallest.
      let indexToSwap = -1;
      if (this.#leftChildIndex(index) >= this.items.length) {
        indexToSwap = this.#rightChildIndex(index);
      } else if (this.#rightChildIndex(index) >= this.items.length) {
        indexToSwap = this.#leftChildIndex(index);
      } else {
        indexToSwap = (this.items[this.#leftChildIndex(index)].priority
          < this.items[this.#rightChildIndex(index)].priority)
          ? this.#leftChildIndex(index) : this.#rightChildIndex(index);
      }

      // Swap with the smallest child.
      const temp = this.items[index];
      this.items[index] = this.items[indexToSwap];
      this.items[indexToSwap] = temp;

      index = indexToSwap;
    }

    return oldRoot;
  }
}
```

Application - Heaps (remove operation)

What is the worst-case running time of the remove operation (n = the heap size)?

Application - Heapsort

```
const pq = new PriorityQueue();
pq.insert("one", 1);
pq.insert("five", 5);
pq.insert("three", 3);
pq.insert("four", 4);
pq.insert("two", 2);

const sorted = [];
while (pq.items.length > 0) {
  sorted.push(pq.remove().item);
}
console.log(sorted);
```

Now we can sort using a heap:

1. Insert everything we want to sort.
2. Repeatedly remove the smallest element.

```
~/projects/sasha-tutorial/2024-02-04 $ node priorityqueue.js
[ 'one', 'two', 'three', 'four', 'five' ]
```

Sidebar - Inverting a binary tree

Common interview question: “invert” a binary tree.

```
for each node n:
    swap n.left and n.right
```

Sidebar - Inverting a binary tree

This is probably not what they're expecting but I guess it works…

```javascript
// Invert a binary tree (annoying answer).
function invert(tree) {
  const initialLength = tree.labels.length;
  for (let i=0; i<initialLength; i++) {
    const tmp = tree.labels[tree.leftChildIndex(i)];
    tree.labels[tree.leftChildIndex(i)] = tree.labels[tree.rightChildIndex(i)];
    tree.labels[tree.rightChildIndex(i)] = tmp;
  }
}
const tree = new BinaryTree(["0", "1", "2", "3", "4"]);
invert(tree);
console.log(tree.labels.filter((cur) => cur !== undefined));
```

?

Sidebar - Inverting a binary tree

```
class Node {
  label;
  left;
  right;

  constructor(label, left, right) {
    this.label = label;
    this.left = left;
    this.right = right;
  }
}

function invert(node) {
  // ???
}
```

Let’s do it for real instead of cheating with our weird interleaved-array representation.

Define a tree node this way instead…

Sidebar - Inverting a binary tree

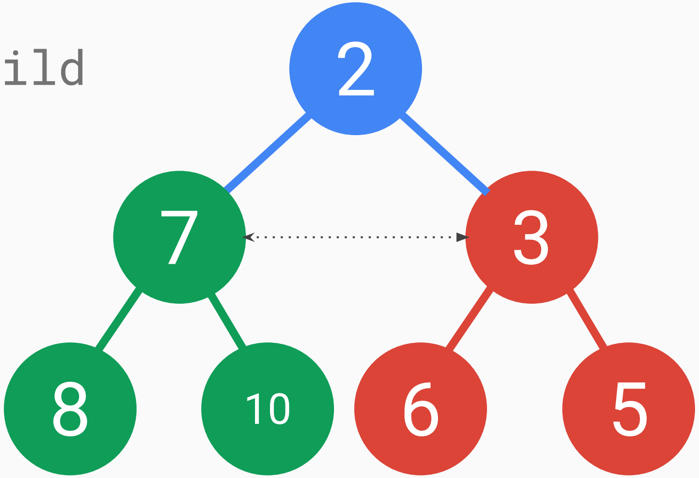
```
class Node {
  label;
  left;
  right;

  constructor(label, left, right) {
    this.label = label;
    this.left = left;
    this.right = right;
  }
}

function invert(node) {
  // ???
}
```

Insight: trees have a **recursive structure**.

The left subtree is itself a tree
rooted at the left child…
The right subtree is itself a tree
rooted at the right child…

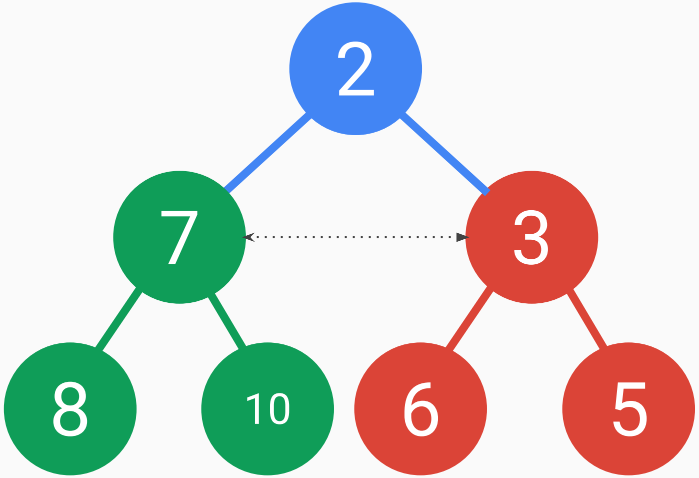
Sidebar - inverting a binary tree

```
to invert a tree:
    invert the left subtree
    invert the right subtree
    swap the left child and right child
```



Sidebar - inverting a binary tree

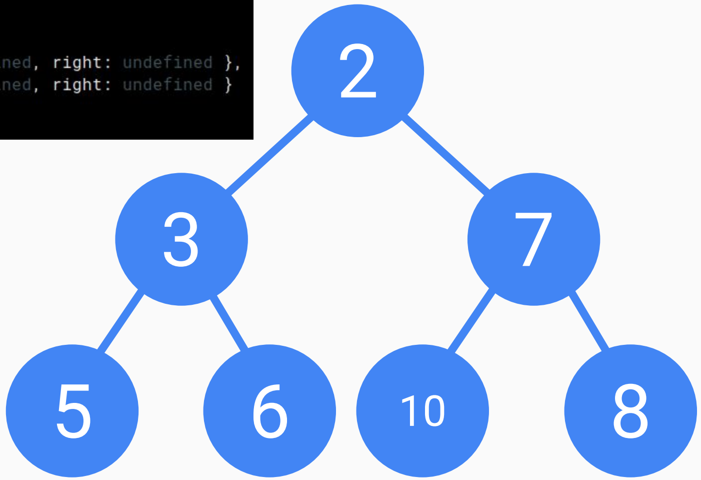
```
function invert(node) {
  if (node.left !== undefined) {
    invert(node.left);
  }

  if (node.right !== undefined) {
    invert(node.right);
  }

  const tmp = node.left;
  node.left = node.right;
  node.right = tmp;
}
```



Sidebar - inverting a binary tree

```
const root = new Node("2",
  new Node("7",
    new Node("8"),
    new Node("10")
  ),
  new Node("3",
    new Node("6"),
    new Node("5")
  )
);
invert(root);
console.log(root);
```

```
~/projects/sasha-tutorial/2024-02-04 $ node binarytree2.js
Node {
  label: '2',
  left: Node {
    label: '3',
    left: Node { label: '5', left: undefined, right: undefined },
    right: Node { label: '6', left: undefined, right: undefined }
  },
  right: Node {
    label: '7',
    left: Node { label: '10', left: undefined, right: undefined },
    right: Node { label: '8', left: undefined, right: undefined }
  }
}
```

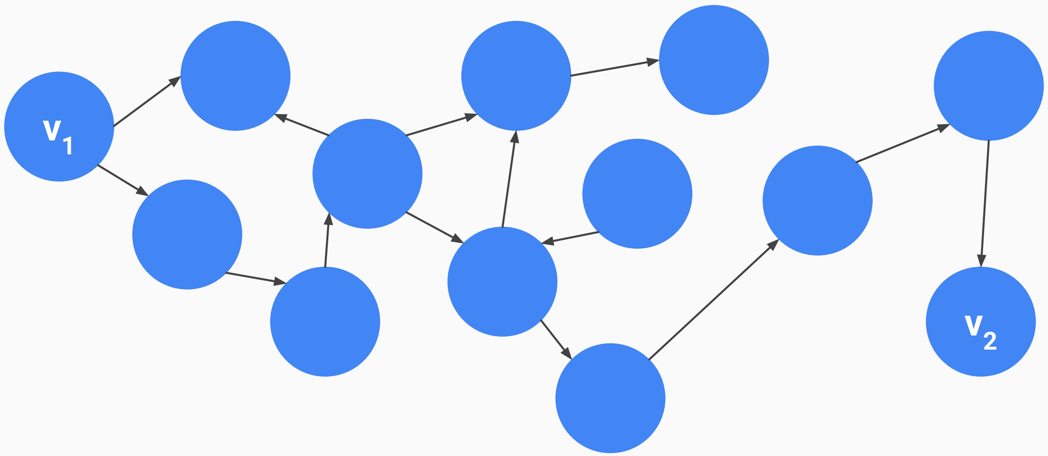


Graphs

Traversals - Searching graphs

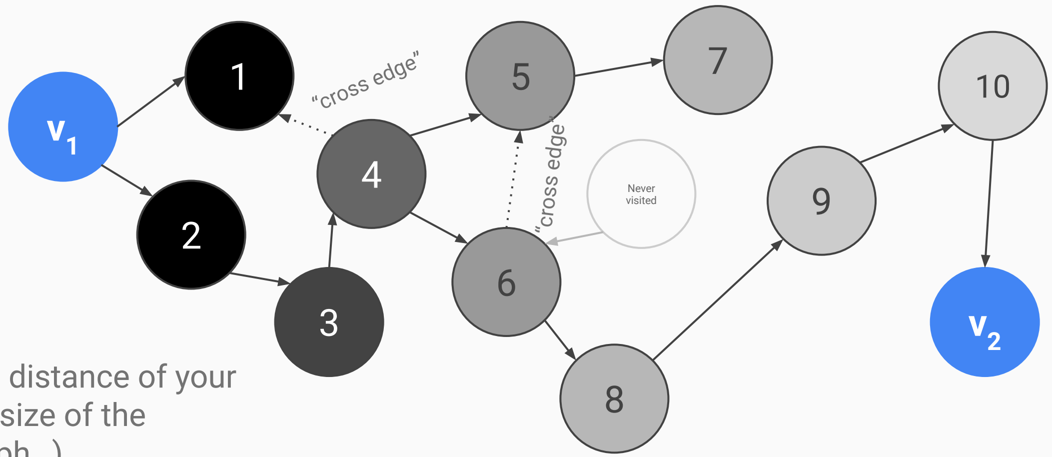
How do we **explore a graph**?

E.g. Given two vertices v_1 and v_2, is v_2 **reachable from** v_1?



Traversals - Breadth first search

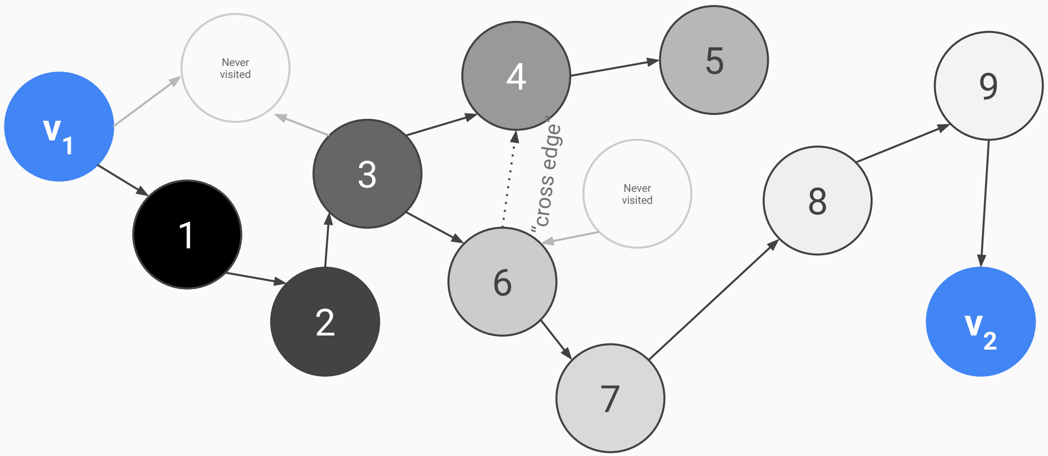
Breadth-first search explores all local vertices, and then all of their neighbors, and then all of their neighbors, and so on. Keep track of visited vertices!



Good for limiting the distance of your search (imagine the size of the Facebook social graph…)

Traversals - Depth first search

Depth-first search goes as deep as it can with one neighbor and then backtracks and tries another neighbor. Keep track of visited vertices!



Traversals - Breadth first search

```
def bfs(start):
    queue := [ start ]
    visited := []
    while queue is not empty:
        dequeue v from the tail of queue
        visit v // e.g. check if it's a match
        add v to visited
        for neighbor in v.neighbors:
            if neighbor not in visited:
                enqueue neighbor at the head of queue
```

Traversals - Depth first search

```
def dfs(start):
    stack := [ start ]
    visited := []
    while stack is not empty:
        pop v from the head of stack
        visit v // e.g. check if it's a match
        add v to visited
        for neighbor in v.neighbors:
            if neighbor not in visited:
                push neighbor at the head of stack
```

Traversals - Depth first search (recursive)

```
def dfs(v, visited):
    visit v // e.g. check if it's a match
    for neighbor in v.neighbors:
        visited ++= dfs(neighbor, visited + v)
    return visited
```

Note that this will visit vertices in the same order as with an explicit stack - **the call stack is our stack!**

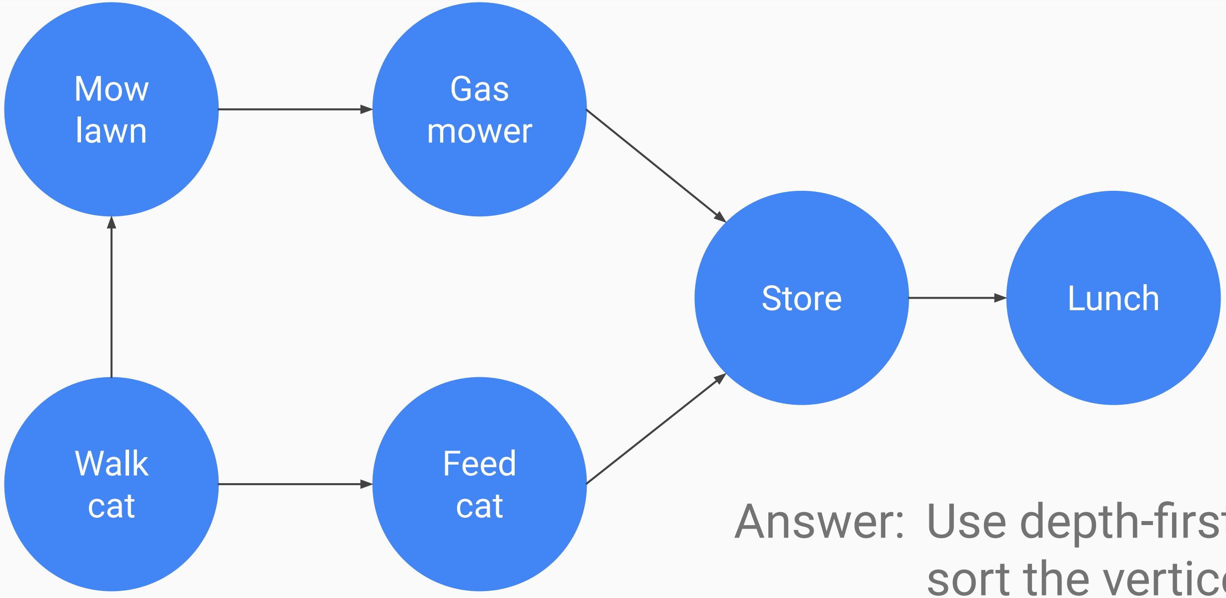
Application - Topological sort

<u>Problem</u>: Given a directed acyclic graph of tasks and their dependencies, put the tasks in a **linear order** so that dependencies are satisfied.

Before I can walk the cat, I have to feed the cat and mow the lawn.
Before I can mow the lawn, I have to gas up the mower.
Before I can gas up the mower I have to go to the store.
Before I can feed the cat I have to go to the store.
Before I can go the the store I have to eat lunch.

⇒ Lunch, store, feed cat, gas mower, mow lawn, walk dog.

Application - Topological sort



Answer: Use depth-first search and sort the vertices by **finish number**.

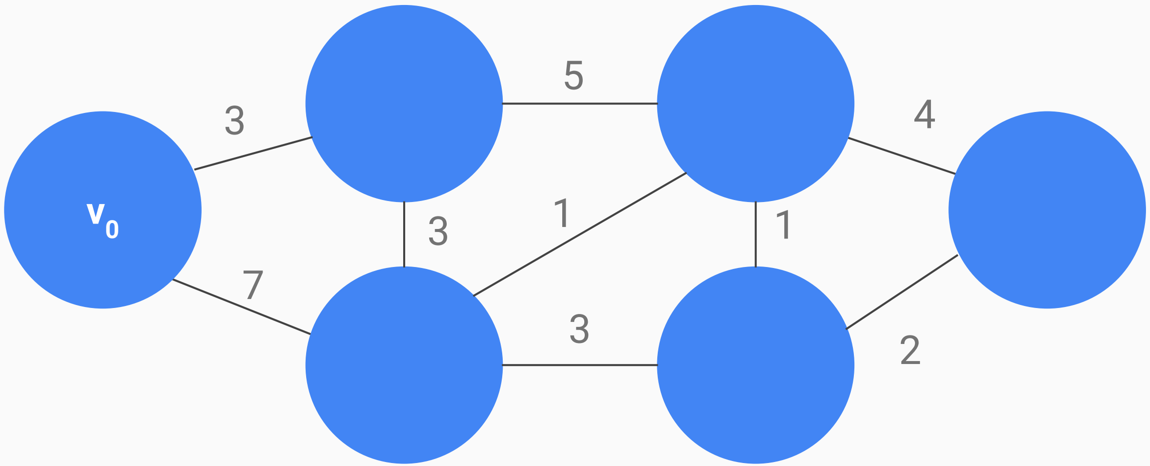
Application - Topological sort

```
def dfs(v, visited):
    for neighbor in v.neighbors:
        visited ++= dfs(neighbor, visited + v)
    finish v
    return visited
```

Note that we've just moved "visit" from the top to be "finish" at the bottom…

Application - Dijkstra's algorithm

Problem: Given a weighted, connected, undirected graph and one vertex $\mathbf{v_0}$ find the shortest paths from $\mathbf{v_0}$ to all other vertices.



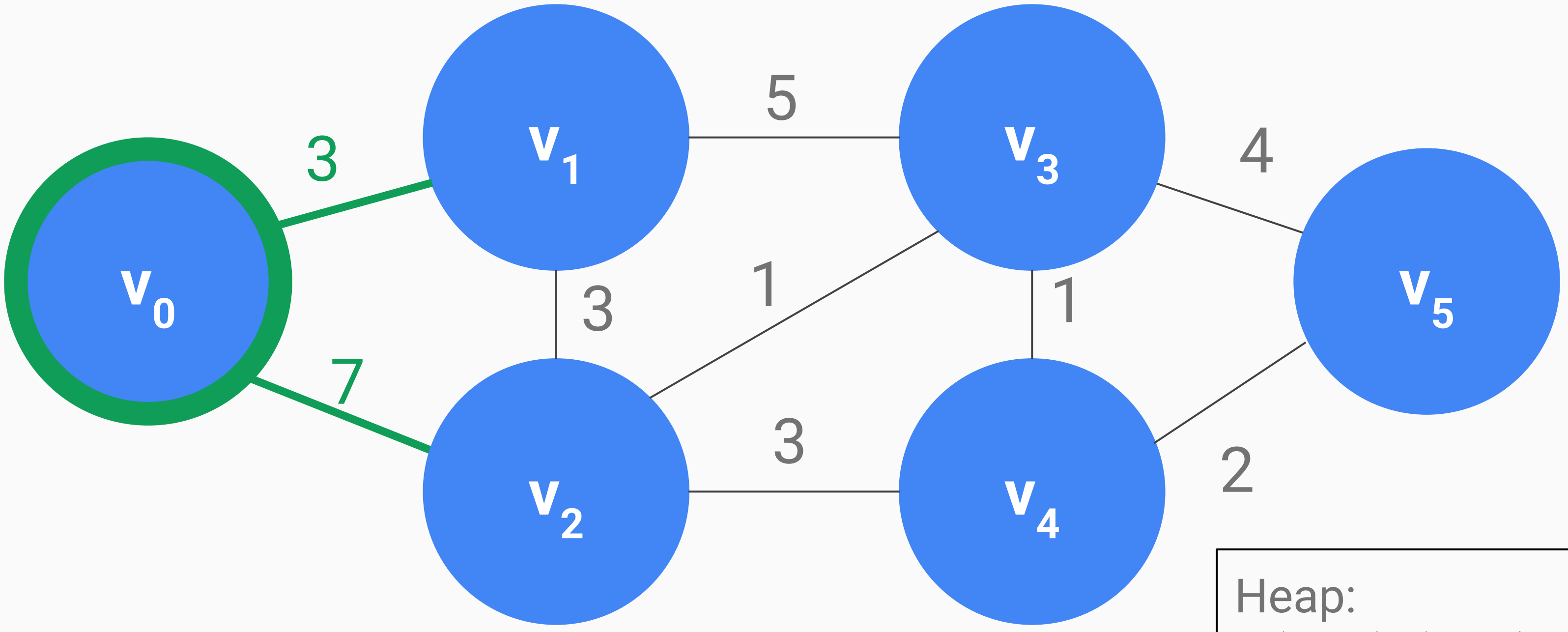
Think of road networks where the weights are `distance` divided by `speedlimit`...

Application - Dijkstra’s algorithm

Ideas:

- Shortest paths will never contain a cycle. (Why?)
- Incrementally build up a set of vertices that are “completed” - i.e. we know we already have the shortest path to them.
- Just follow the **smallest edge** from a “completed” vertex to an “uncompleted” vertex in order to expand our “completed” set.
- Use a heap to keep track of the “frontier!”

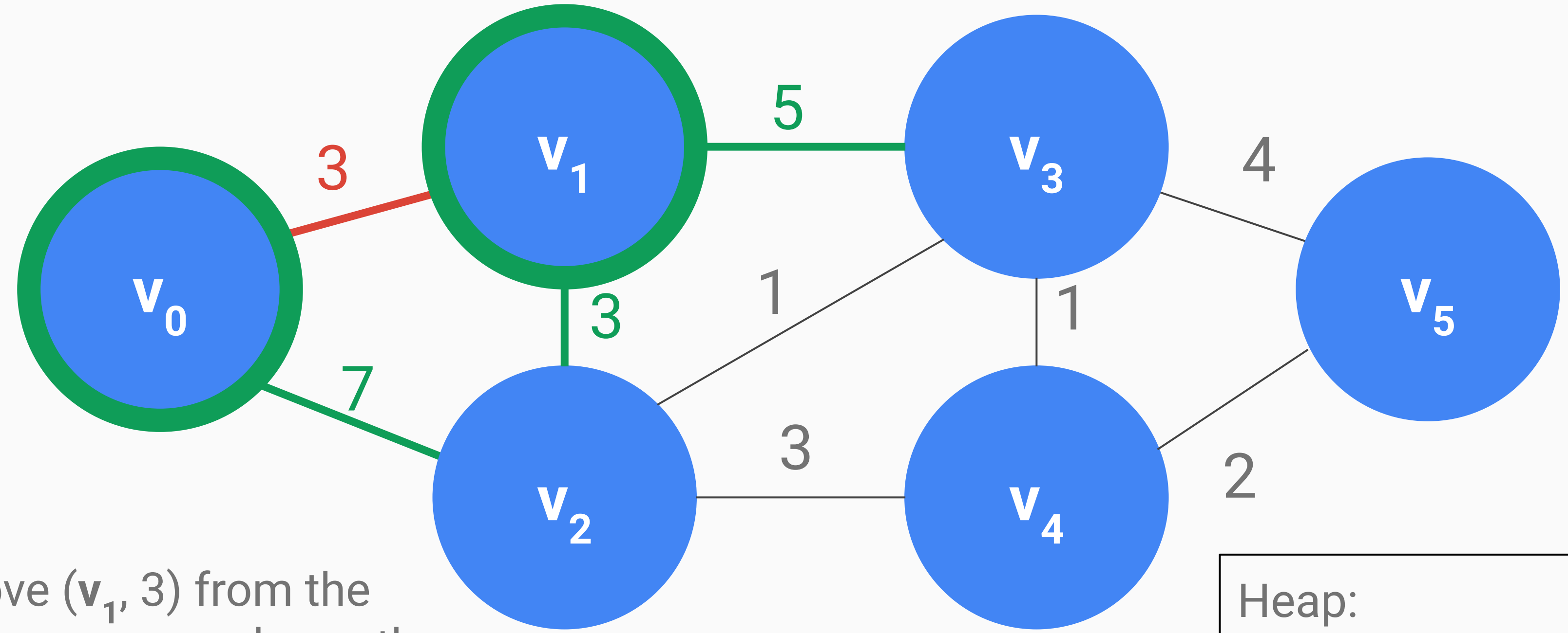
Application - Dijkstra's algorithm



Heap:
($\mathbf{v}_1$, 3), ($\mathbf{v}_2$, 7)
Completed:
$\mathbf{v}_0$

Add ($\mathbf{v}_1$, 3) and ($\mathbf{v}_2$, 7) to the heap.

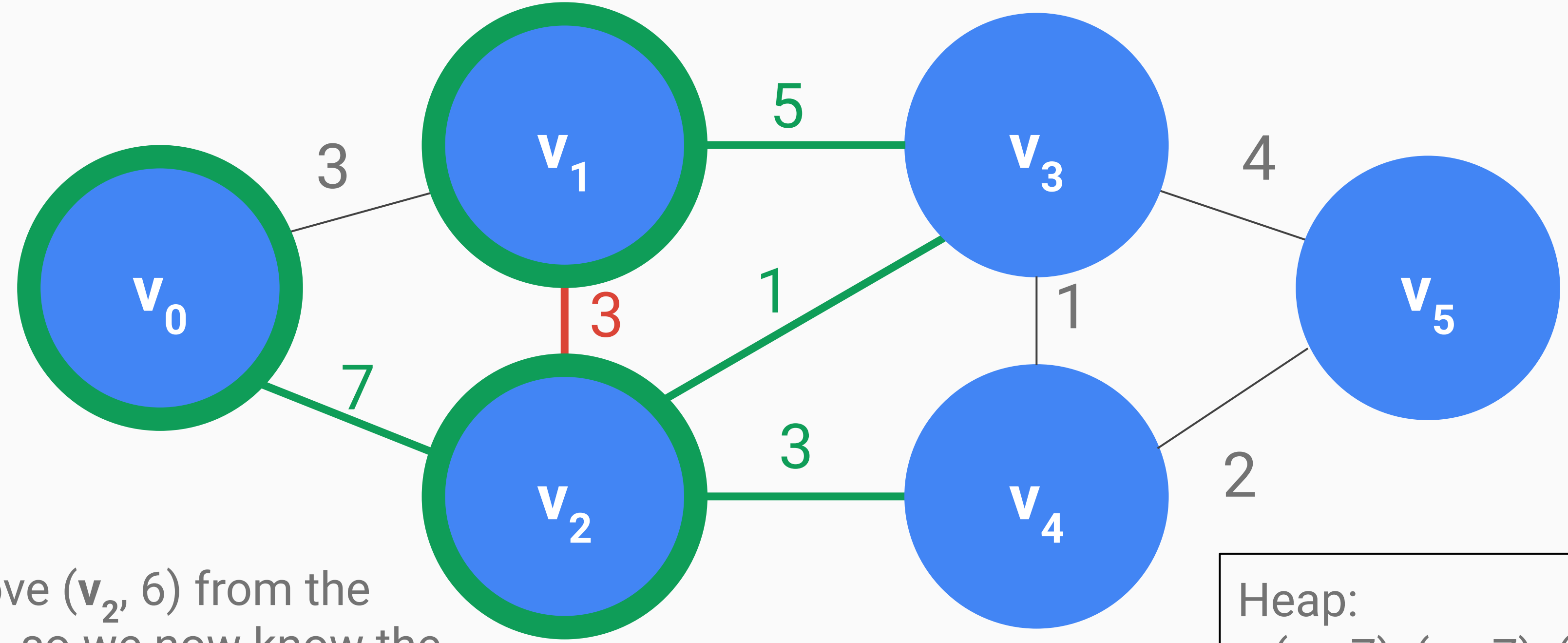
Application - Dijkstra's algorithm



Remove ($\mathbf{v}_1$, 3) from the heap - so we now know the shortest path to $\mathbf{v}_1$. Look at its neighbors and add ($\mathbf{v}_3$, 8) and ($\mathbf{v}_2$, 6) to the heap.

Heap:
($\mathbf{v}_2$, 6), ($\mathbf{v}_2$, 7), ($\mathbf{v}_3$, 8)
Completed:
$\mathbf{v}_0$, $\mathbf{v}_1$

Application - Dijkstra's algorithm

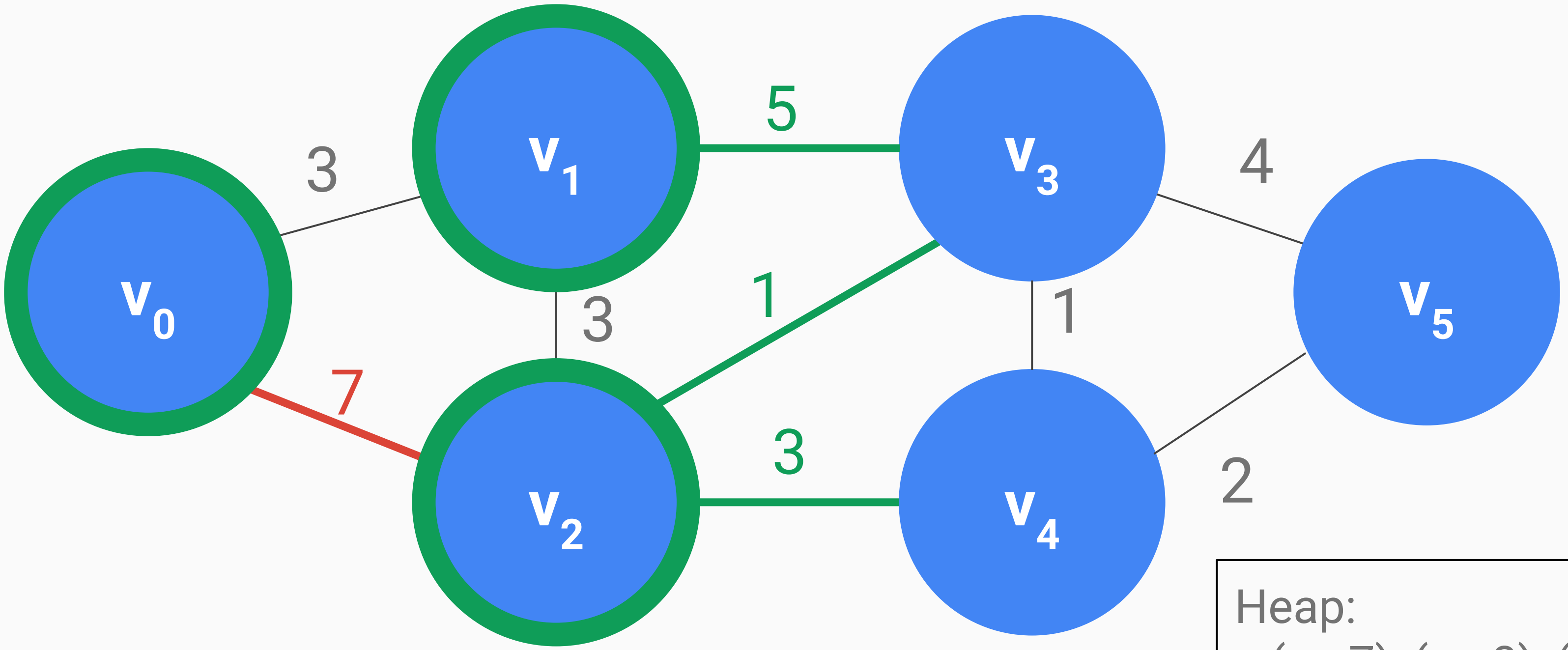


Remove ($\mathbf{v}_2$, 6) from the heap - so we now know the shortest path to $\mathbf{v}_2$. Look at its neighbors and add ($\mathbf{v}_3$, 7) and ($\mathbf{v}_4$, 9) to the heap.

Heap:
($\mathbf{v}_2$, 7), ($\mathbf{v}_3$, 7), ($\mathbf{v}_3$, 8), ($\mathbf{v}_4$, 9)

Completed:
$\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$

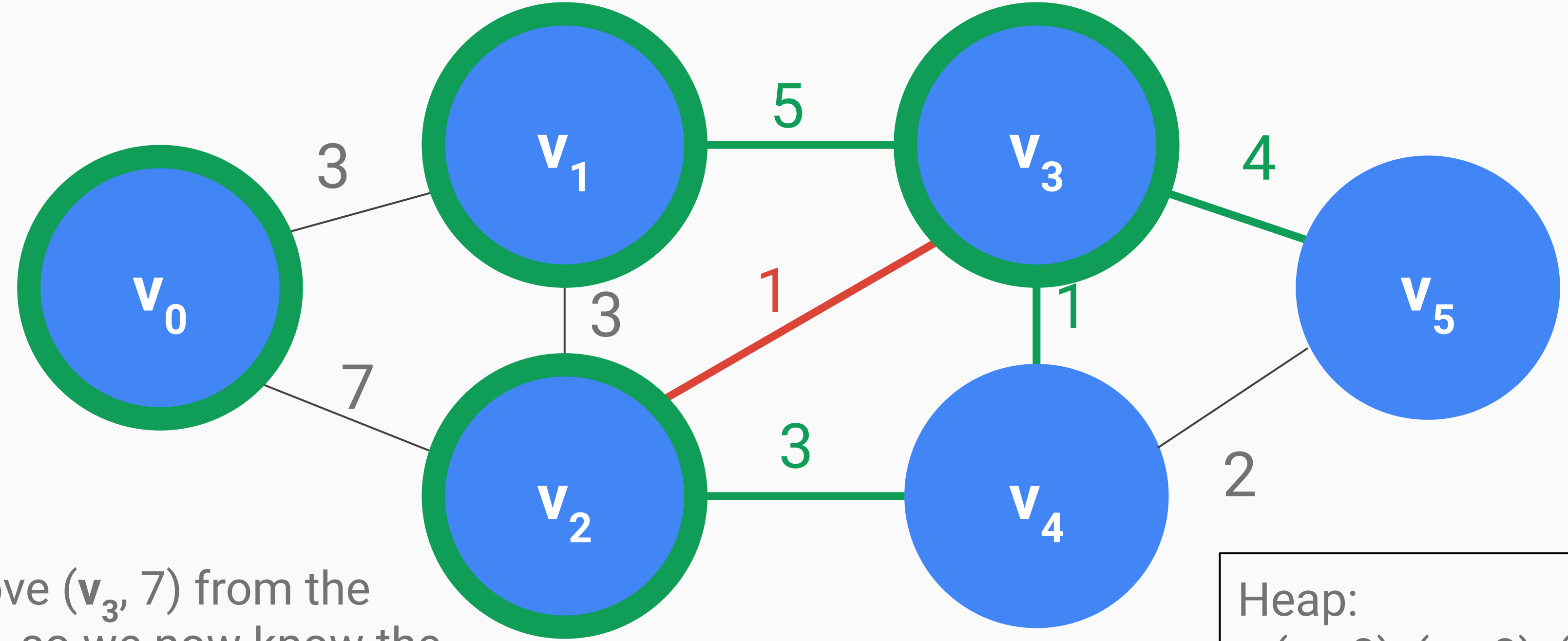
Application - Dijkstra's algorithm



Remove ($\mathbf{v}_2$, 7) from the heap - but we've already completed $\mathbf{v}_2$ so do nothing.

Heap:
($\mathbf{v}_3$, 7), ($\mathbf{v}_3$, 8), ($\mathbf{v}_4$, 9)
Completed:
$\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$

Application - Dijkstra's algorithm

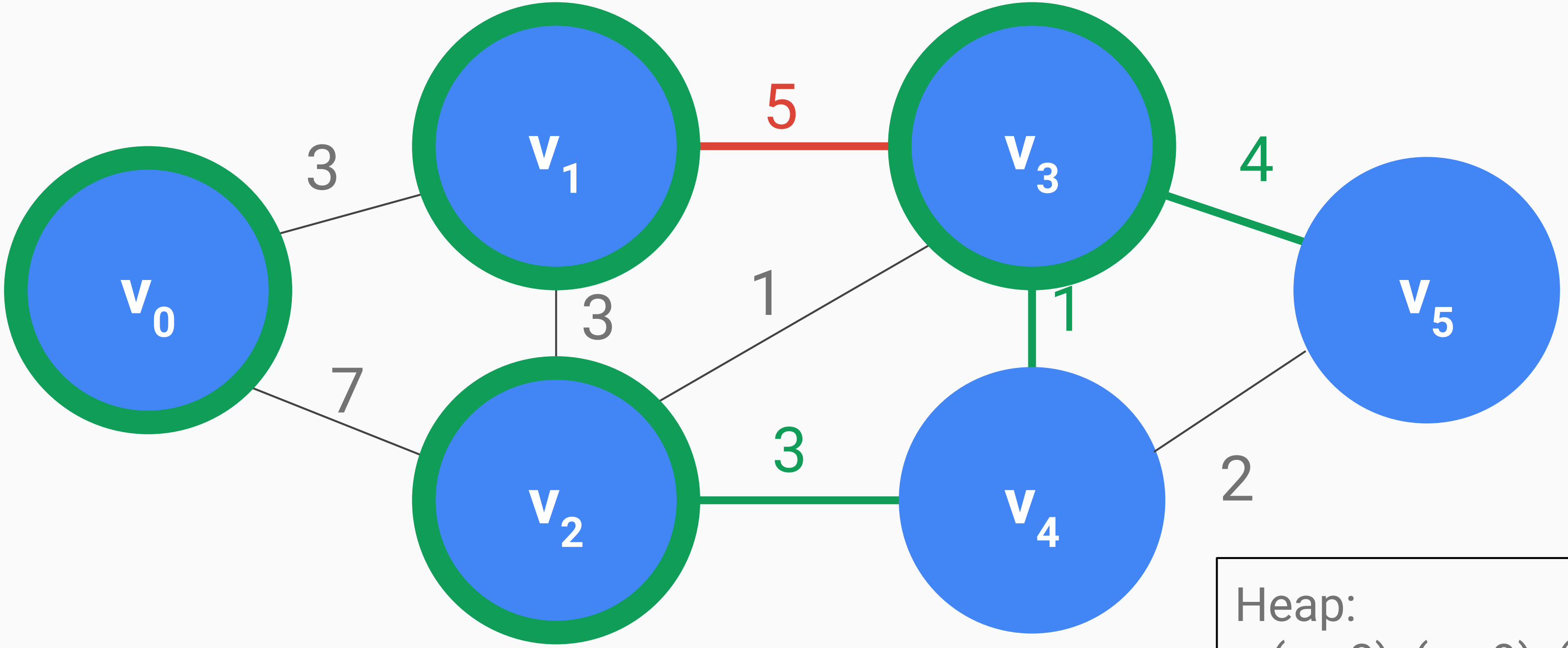


Remove ($\mathbf{v}_3$, 7) from the heap - so we now know the shortest path to $\mathbf{v}_3$. Look at its neighbors and add ($\mathbf{v}_4$, 8) and ($\mathbf{v}_5$, 11) to the heap.

Heap:
($\mathbf{v}_3$, 8), ($\mathbf{v}_4$, 8), ($\mathbf{v}_4$, 9), ($\mathbf{v}_5$, 11)

Completed:
$\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$

Application - Dijkstra's algorithm

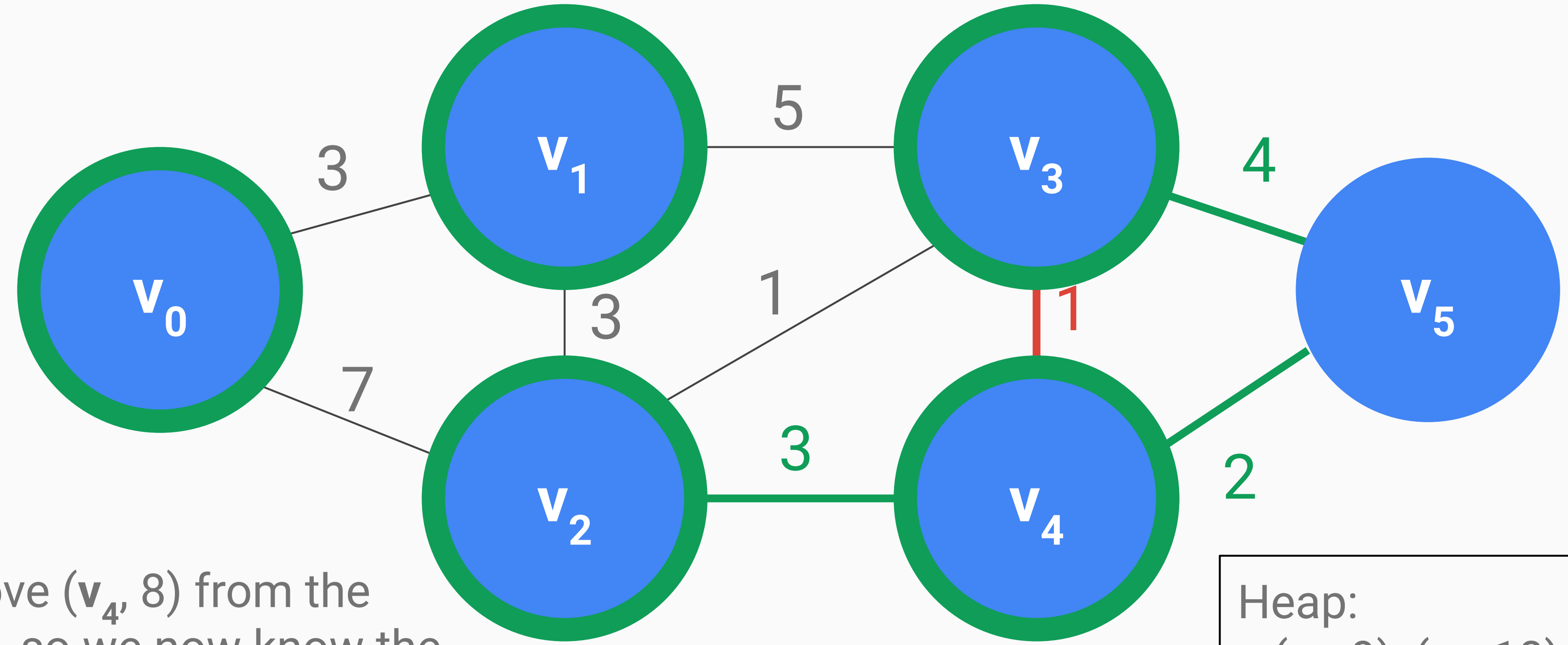


Remove ($\mathbf{v}_3$, 8) from the heap - but we've already completed $\mathbf{v}_3$ so do nothing.

Heap:
($\mathbf{v}_4$, 8), ($\mathbf{v}_4$, 9), ($\mathbf{v}_5$, 11)

Completed:
$\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$

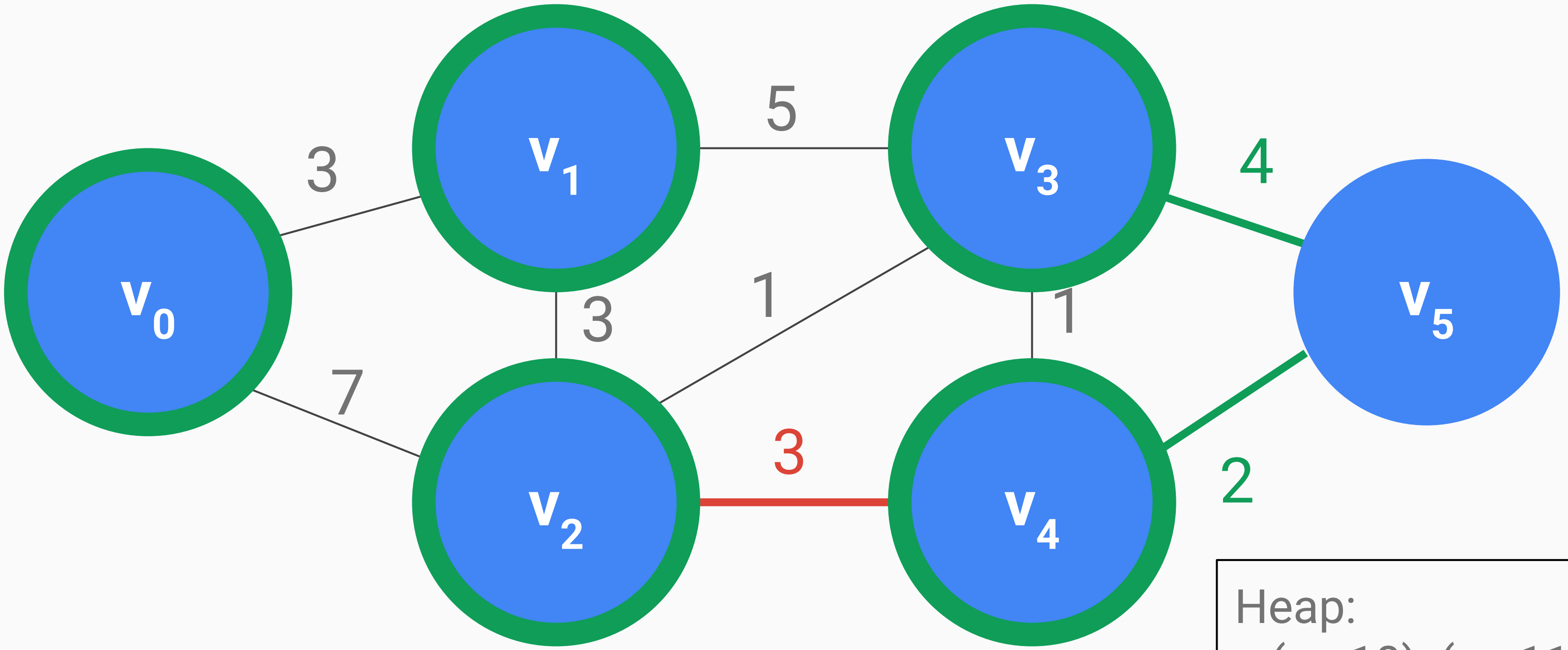
Application - Dijkstra's algorithm



Remove (**v_4**, 8) from the heap - so we now know the shortest path to **v_4**. Look at its neighbors and add (**v_5**, 10) to the heap.

Heap:
(**v_4**, 9), (**v_5**, 10), (**v_5**, 11)
Completed:
v_0, **v_1**, **v_2**, **v_3**, **v_4**

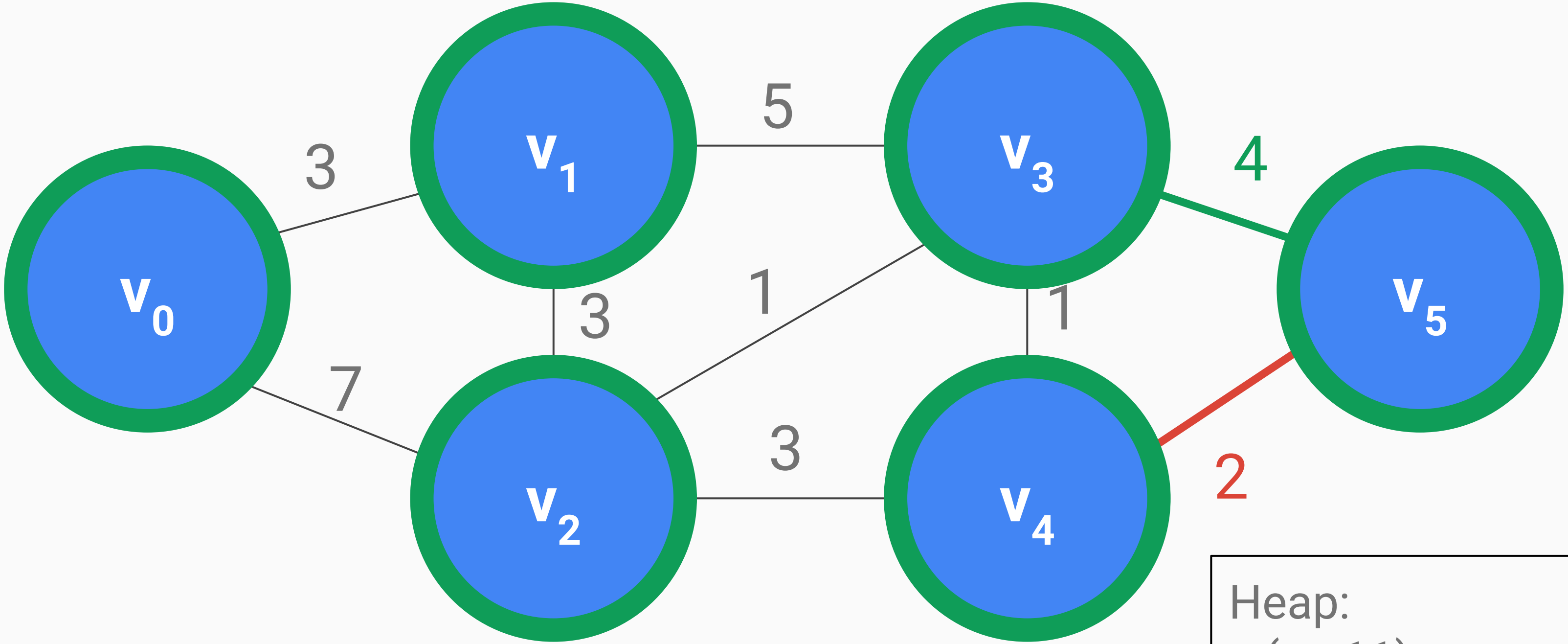
Application - Dijkstra's algorithm



Remove ($\mathbf{v}_4$, 9) from the heap - but we've already completed $\mathbf{v}_4$ so do nothing.

Heap:
($\mathbf{v}_5$, 10), ($\mathbf{v}_5$, 11)
Completed:
$\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$, $\mathbf{v}_4$

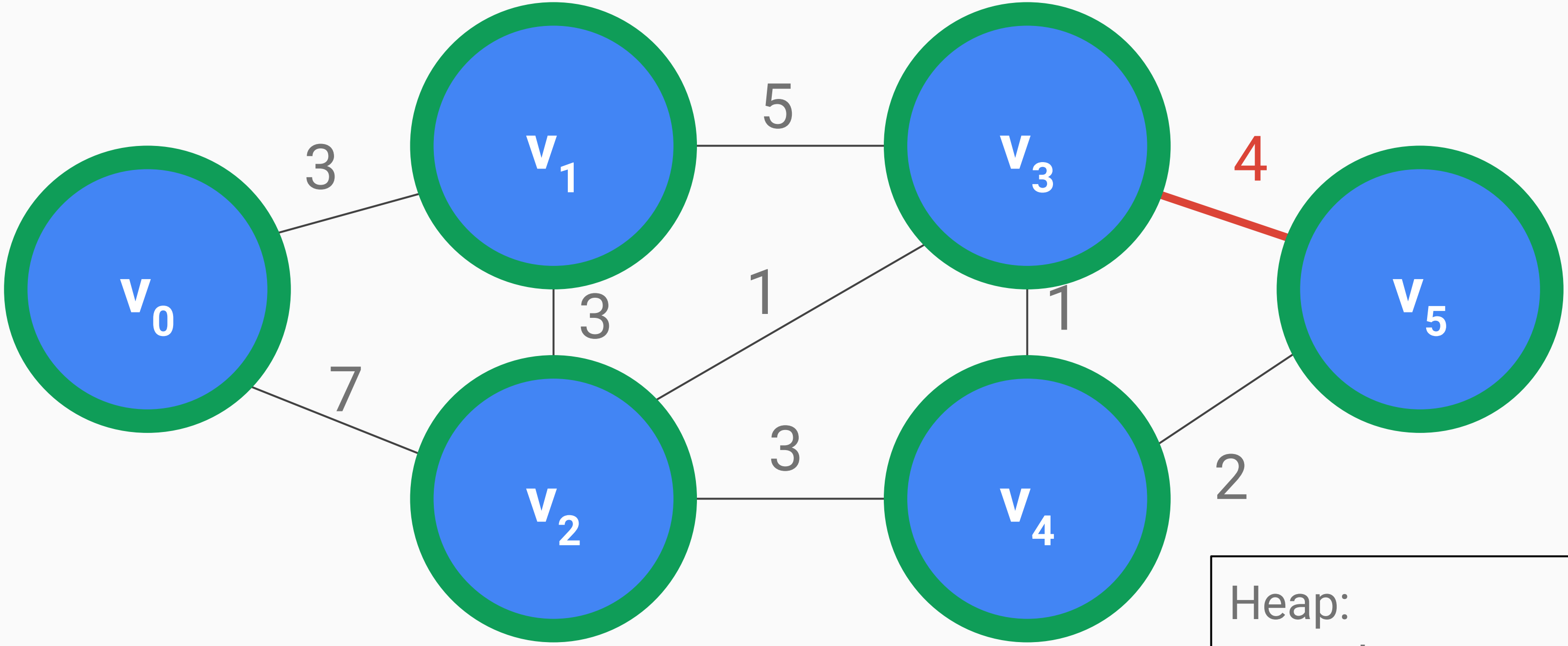
Application - Dijkstra’s algorithm



Remove ($\mathbf{v}_5$, 10) from the heap - so we now know the shortest path to $\mathbf{v}_5$.

Heap:
($\mathbf{v}_5$, 11)
Completed:
$\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$, $\mathbf{v}_4$, $\mathbf{v}_5$

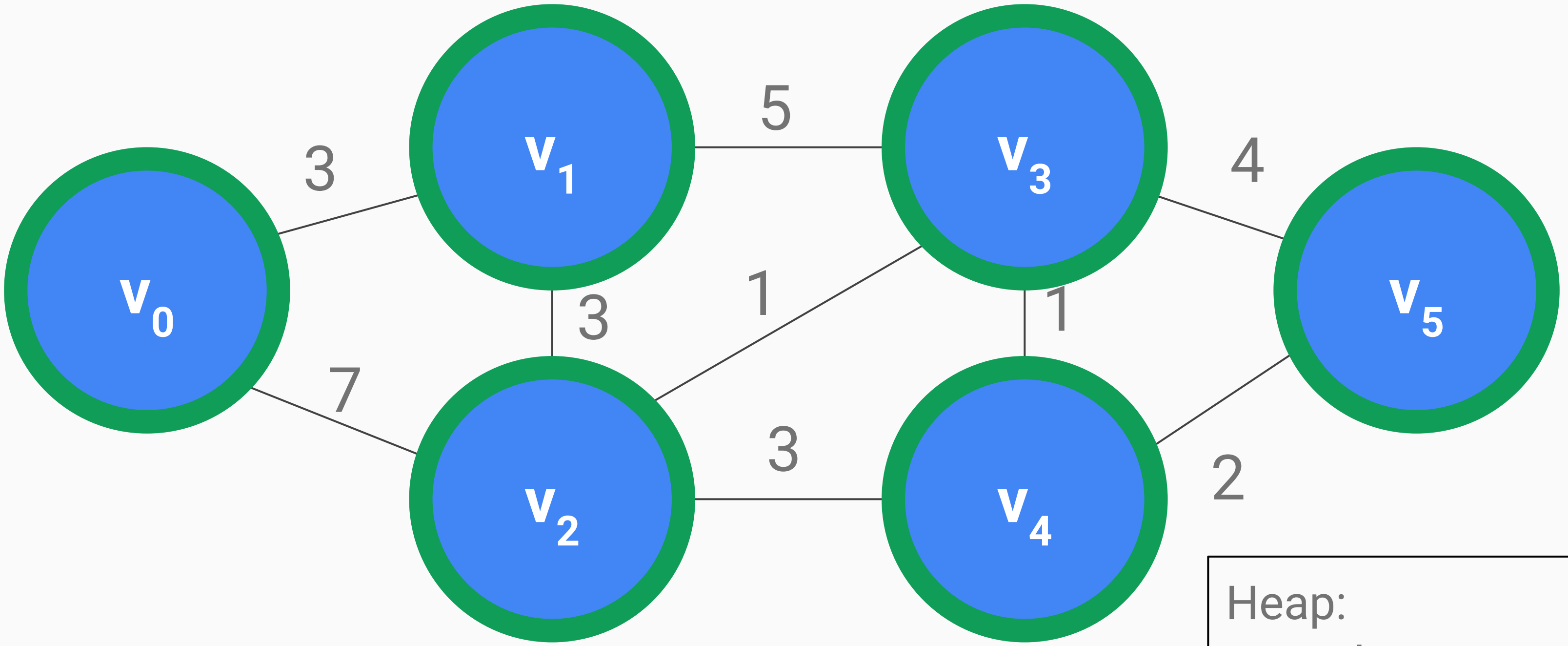
Application - Dijkstra's algorithm



Remove (**v_5**, 11) from the heap - but we've already completed **v_5** so do nothing.

Heap:
empty
Completed:
v_0, v_1, v_2, v_3, v_4, v_5

Application - Dijkstra's algorithm



Heap:
empty
Completed:
$v_0, v_1, v_2, v_3, v_4, v_5$

There's nothing left in the heap, so we're done!